JMLR: Workshop and Conference Proceedings 34:79–93, 2014 Proceedings of the 12th ICGI

An Abstract Framework for Counterexample Analysis in Active Automata Learning

Malte Isberner MALTE.ISBERNER@TU-DORTMUND.DE
Bernhard Steffen STEFFEN@CS.TU-DORTMUND.DE
TU Dortmund, Department of Computer Science, Chair for Programming Systems
D-44227 Dortmund, Germany

Editors: Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka

Abstract

Counterexample analysis has emerged as one of the key challenges in Angluin-style active automata learning. Rivest and Schapire (1993) showed for the L* algorithm that a single suffix of the counterexample was sufficient to ensure progress. This suffix can be obtained in a binary search fashion, requiring $\Theta(\log m)$ membership queries for a counterexample of length m. Correctly implementing this algorithm can be quite tricky, and its correctness sometimes even has been disputed. In this paper, we establish an abstract framework for counterexample analysis, which basically reduces the problem of finding a suffix to finding distinct neighboring elements in a 0/1 sequence, where the first element is 0 and the last element is 1. We demonstrate the conciseness and simplicity of our framework by using it to present new counterexample analysis algorithms, which, while maintaining the worst-case complexity of $O(\log m)$, perform significantly better in practice. Furthermore, we contribute—in a second instantiation of our framework, highlighting its generality—the first sublinear counterexample analysis procedures for the algorithm due to Kearns and Vazirani (1994).

Keywords: Automata learning, counterexamples, Rivest&Schapire, binary search, exponential search

1. Introduction

When Angluin (1987) presented her famous L* algorithm for inferring regular languages, a novelty was not only the algorithm as such, but also the formalization of a *Minimally Adequate Teacher* (MAT). Besides answering queries regarding the language membership of words, the MAT—or, more precisely, an *equivalence query*—also provides the learner with *counterexamples* once a hypothesis is conjectured. However, the way these counterexamples were handled was not very sophisticated: in the original version of L*, all prefixes of a counterexample were added to the observation table. This has several drawbacks: first, "unproductive" prefixes of the counterexample can account for a significant portion of all membership queries. Second, it does not make explicit where exactly the hypothesis is erroneous. Third, it introduced the phenomenon of *inconsistencies*, a symptom of forgoing *uniqueness of representatives* of states in the hypothesis.

Rivest and Schapire (1993) showed that it was possible to add a single distinguishing suffix to ensure progress, and that furthermore this suffix is contained in the counterexample and can be found efficiently using binary search. Extracting such a suffix thus requires only

$\lceil \log m \rceil$ membership queries for a counterexample of length m. However, the correctness of the search algorithm is not at all obvious, and implementing it correctly requires avoiding several pitfalls. These aspects have even led to its correctness being disputed (Irfan, 2012).

Another problem of Rivest&Schapire's algorithm concerns its practical performance: while $\lceil \log m \rceil$ is a relatively small number even for larger values of m, many heuristics—while possibly suffering from a linear worst-case query complexity—frequently perform better, yield shorter suffixes, or both.

In this paper, we address all of the above aspects. Essential for this is a new, simplified perspective on counterexample analysis. Our main contributions are the following:

- a framework that facilitates both understanding and implementing counterexample analysis algorithms based on finding suffixes,
- new analysis algorithms that maintain Rivest&Schapire's upper bound of $O(\log m)$ queries, but may perform better in practice,
- another instance of the framework for the algorithm by Kearns and Vazirani (1994), yielding sublinear counterexample analysis procedures for this algorithm, and
- an experimental comparison of these algorithms on randomly generated automata, focusing on the cost of counterexample analysis.

Related Work. Active Automata Learning in its modern form can be attributed to Angluin (1987), who established the MAT framework with its membership and equivalence queries, as well as L*, the first (and best-known) active automata learning algorithm formulated in the MAT framework. Rivest and Schapire (1993) proposed the above-mentioned modification to L*, i.e., adding only a single distinguishing suffix found using binary search. Irfan et al. (2010) have proposed other modifications to the counterexample handling of L*, motivated by the problem of non-optimal (i.e., long) counterexamples that result from equivalence query approximations. Despite performing good in practice, these heuristics have a linear worst-case complexity, and moreover are only applicable to observation table-based learning algorithms. Howar (2012) combined Rivest&Schapire's counterexample analysis algorithm with the *discrimination tree* data structure introduced by Kearns and Vazirani (1994), forming the *Observation Pack* algorithm. We will use this algorithm for our evaluation, exploiting the fact that the counterexample analysis is interchangeable, as long as the analysis algorithms satisfies certain properties.

Outline. The paper is structured as follows. Section 2 establishes the mathematical notation and the formalization of DFA and related concepts that are used throughout the paper. Section 3 revisits active automata learning, using the Observation Pack algorithm as an example, particularly highlighting the role of counterexample analysis. In Section 4, we establish our abstract framework for counterexample analysis, which is used to discuss further optimization and alternative algorithms in Section 5. In Section 6, we demonstrate how the framework can be instantiated for an algorithm of a different flavor, namely the one due to Kearns and Vazirani (1994). Section 7 reports on the results of our experimental evaluation, before Section 8 concludes the paper.

2. Preliminaries

2.1. Mathematical Notation

Let $\mathbb{N}$ denote the set of all natural numbers, including 0. For $n, m \in \mathbb{N}$, we write $[n, m) = \{i \in \mathbb{N} \mid n \leq i < m\}$ to denote the half-open integer interval containing all natural numbers between n (inclusive) and m (exclusive).[1] We identify the Boolean truth values *true* and *false* with 1 and 0, respectively; i.e., $\mathbb{B} = \{0, 1\}$.

2.2. Alphabets, Words, and Deterministic Finite Automata

Let Σ be a finite set of symbols (the "input alphabet"). A *word* w over Σ is a finite sequence of symbols ranging over Σ. The set of all words is denoted by Σ^*, and includes the empty word $\boldsymbol{\varepsilon}$ (i.e., the unique word of length 0). The set of all words of positive length is denoted by Σ^+, and we have $\Sigma^+ = \Sigma^* \setminus \{\boldsymbol{\varepsilon}\}$.

For a word $w \in \Sigma^*$, we denote its length by $|w|$. The single symbols constituting the word are usually referred to by w_i, $0 \leq i < |w|$; thus, $w = w_0 w_1 \dots w_{|w|-1}$. For some integer interval $I \subseteq [0, |w|)$, w_I is the word resulting from taking into account only the indices contained in I. Hence, $w_{[0,k)}$ is the *prefix of* w of length k, and $w_{[k,|w|)}$ is the *suffix of* w starting at index k. Note that $w_{[0,0)} = w_{[|w|,|w|)} = \boldsymbol{\varepsilon}$ and $w_{[0,|w|)} = w$.

The concatenation of words $w, w' \in \Sigma^*$ is denoted by $w \cdot w'$, or simply ww'. We use the former notation whenever we want to emphasize the logical subdivision a concatenation operation reflects.

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

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

The intuition behind a DFA is the following: at every point in time, $\mathcal{A}$ is in some state $q \in Q^{\mathcal{A}}$, starting with the initial state $q_0^{\mathcal{A}}$. Upon reading a symbol $a \in \Sigma$, it moves to a *successor state* $\delta^{\mathcal{A}}(q, a)$, according to the transition function $\delta^{\mathcal{A}}$.

A word $w \in \Sigma^*$ is *accepted* by a DFA $\mathcal{A}$ iff, after processing all the symbols in w from left to right, $\mathcal{A}$ is in an accepting state (i.e., a state $q \in F^{\mathcal{A}}$). We formalize this description by extending the transition function $\delta^{\mathcal{A}}$ from symbols to words. We define:

$$\delta^{\mathcal{A}}(q, \boldsymbol{\varepsilon}) = q, \quad \delta^{\mathcal{A}}(q, w \cdot a) = \delta^{\mathcal{A}}(\delta^{\mathcal{A}}(q, w), a) \text{ for all } q \in Q^{\mathcal{A}}, w \in \Sigma^*, a \in \Sigma.$$

For the extended transition function originating from the initial state $q_0^{\mathcal{A}}$, we use a notation borrowed from Kearns and Vazirani (1994), defining $\mathcal{A}[w] = \delta^{\mathcal{A}}(q_0^{\mathcal{A}}, w)$. A word $w \in \Sigma^*$ is thus accepted by $\mathcal{A}$ iff $\mathcal{A}[w] \in F^{\mathcal{A}}$. We naturally extend this notation to sets of words $W \subseteq \Sigma^*$: $\mathcal{A}[W] = \{\mathcal{A}[w] \mid w \in W\}$.

1. To avoid confusion, we will only consider half-open intervals in this paper. Thus, we will generally favor $[n, m+1)$ over $[n, m]$. Note that $[0, n)$ coincides with the index range of a C/Java array of length n.

In the context of active automata learning, it is usually more convenient to regard acceptance as an *output function*. Let $\lambda_q^{\mathcal{A}} \colon \Sigma^* \to \mathbb{B}$ be the output function for state $q \in Q^{\mathcal{A}}$, defined by $\lambda_q^{\mathcal{A}}(w) = 1$ iff $\delta^{\mathcal{A}}(q, w) \in F^{\mathcal{A}}$. We use $\lambda^{\mathcal{A}}$ as shorthand for $\lambda_{q_0^{\mathcal{A}}}^{\mathcal{A}}$. Obviously, $\lambda^{\mathcal{A}}(w) = 1$ iff $\mathcal{A}[w] \in F^{\mathcal{A}}$. We define two DFA $\mathcal{A}$, $\mathcal{A}'$ to be *equivalent*, denoted $\mathcal{A} \cong \mathcal{A}'$, iff they have the same output function, i.e., iff $\lambda^{\mathcal{A}} = \lambda^{\mathcal{A}'}$.

Definition 2 (Isomorphy of DFA) *Let $\mathcal{A}$ and $\mathcal{A}'$ be DFA over Σ. $\mathcal{A}$ and $\mathcal{A}'$ are called* isomorphic *iff there exists a bijection $f \colon Q^{\mathcal{A}} \to Q^{\mathcal{A}'}$ satisfying the following conditions:*

1. $f(q_0^{\mathcal{A}}) = q_0^{\mathcal{A}'}$
2. $\forall q \in Q^{\mathcal{A}} : q \in F^{\mathcal{A}} \Leftrightarrow f(q) \in F^{\mathcal{A}'}$
3. $\forall q \in Q^{\mathcal{A}}, a \in \Sigma : f(\delta^{\mathcal{A}}(q, a)) = \delta^{\mathcal{A}'}(f(q), a)$

In this case, f is also called an isomorphism*.*

Isomorphy is a stronger requirement than equivalence: isomorphic DFA are also equivalent, but in general not vice versa.

Definition 3 (Canonical DFA) *Let $\mathcal{A}$ be a DFA over Σ. $\mathcal{A}$ is called* canonical *iff the following holds:*

1. *All states are* reachable*: $\mathcal{A}[\Sigma^*] = Q^{\mathcal{A}}$.*
2. *All states are pairwisely* separable*: $\forall q \neq q' \in Q^{\mathcal{A}} : \exists w \in \Sigma^* : \lambda_q^{\mathcal{A}}(w) \neq \lambda_{q'}^{\mathcal{A}}(w)$.*

Intuitively speaking, a canonical DFA $\mathcal{A}$ does not contain any *redundant* states. For any DFA $\mathcal{A}$, there exists exactly one (up to isomorphism) *canonical* DFA $\hat{\mathcal{A}}$ satisfying $\mathcal{A} \cong \hat{\mathcal{A}}$ (Nerode, 1958).

3. Angluin-style DFA Learning

Active automata learning is the problem of identifying (up to equivalence) an unknown ("target") DFA $\mathcal{A}$ over a given alphabet Σ.[2] To this end, the "learner" may pose two kinds of queries to a "teacher": a *membership query* for a word $w \in \Sigma^*$ corresponds to evaluating $\lambda^{\mathcal{A}}(w)$. If the learner has conjectured a hypothesis DFA $\mathcal{H}$, it may pose an *equivalence query*, asking if $\mathcal{H} \cong \mathcal{A}$. If this is the case, *ok* is returned, and learning can terminate. Otherwise, a *counterexample* is returned, i.e., a word $w \in \Sigma^*$ satisfying $\lambda^{\mathcal{A}}(w) \neq \lambda^{\mathcal{H}}(w)$. A teacher capable of answering these queries is called a *minimally adequate teacher* (MAT). Angluin (1987) showed that in the presence of a MAT, $\mathcal{A}$ can be learned using polynomially many membership and linearly many equivalence queries, where both the target DFA as well as the counterexamples are treated as inputs (thus, "polynomially" means polynomially in both the number of states of the target DFA and the length of the longest counterexample). Angluin also presented L*, the first algorithm accomplishing this task.

2. Without loss of generality, we assume $\mathcal{A}$ to be canonical

3.1. Realization: The Observation Pack Algorithm

Despite the popularity of L*, we choose the Observation Pack algorithm, due to Howar (2012), to illustrate the key concepts of active learning. However, a large amount of the following description also applies to other active automata learning algorithms, such as L*.

The Observation Pack algorithm uses a finite, prefix-closed set $\mathcal{S}p \subset \Sigma^*$ to identify states in both $\mathcal{A}$ and the hypothesis $\mathcal{H}$. Every state $q \in Q^{\mathcal{H}}$ uniquely corresponds to a word (called *short prefix*) $u \in \mathcal{S}p$, and it is ensured that $\mathcal{H}[u] = q$. We call u the *access sequence* of q (wrt. $\mathcal{H}$), denoted $\lfloor q \rfloor_{\mathcal{H}}$. The above condition can thus be formulated as $\forall q \in Q^{\mathcal{H}} : \mathcal{H}[\lfloor q \rfloor_{\mathcal{H}}] = q$. The initial state $q_0^{\mathcal{H}}$ of $\mathcal{H}$ is the state with access sequence ε.

We extend the access sequence notation to arbitrary words $w \in \Sigma^*$: $\lfloor w \rfloor_{\mathcal{H}} = \lfloor \mathcal{H}[w] \rfloor_{\mathcal{H}}$. Hence, the mapping $\lfloor \cdot \rfloor_{\mathcal{H}} : \Sigma^* \to \mathcal{S}p$ *transforms* words into access sequences.

A short prefix $u \in \mathcal{S}p$ naturally corresponds to a state in $\mathcal{A}$, namely $\mathcal{A}[u]$. We refer to $\mathcal{A}[\mathcal{S}p]$ as the *discovered* states of $\mathcal{A}$. Short prefixes thus establish a mapping $f_{\mathcal{S}p}$ between states in the hypothesis and discovered states in the target DFA, defined as follows:

$$f_{\mathcal{S}p} \colon Q^{\mathcal{H}} \to Q^{\mathcal{A}}, \quad f_{\mathcal{S}p}(q) = \mathcal{A}\left[\lfloor q \rfloor_{\mathcal{H}}\right].$$

Note that $f_{\mathcal{S}p}(q_0^{\mathcal{H}}) = f_{\mathcal{S}p}(\mathcal{H}[\varepsilon]) = \mathcal{A}(\varepsilon) = q_0^{\mathcal{A}}$. This mapping lays the foundation for the following three *learning invariants*, maintaining of which guarantees correctness upon termination:

(I1) Different short prefixes $u \neq u' \in \mathcal{S}p$ correspond to different states in the target DFA: $\mathcal{A}[u] \neq \mathcal{A}[u']$. In other words, $f_{\mathcal{S}p}$ is an *injection*.

(I2) Acceptances of identified states are correct in $\mathcal{H}$: $\forall q \in Q^{\mathcal{H}} : q \in F^{\mathcal{H}} \Leftrightarrow f_{\mathcal{S}p}(q) \in F^{\mathcal{A}}$.

(I3) Transitions in $\mathcal{H}$ point to the correct target state, if the latter has already been discovered: $\forall q \in Q^{\mathcal{H}}, a \in \Sigma : \delta^{\mathcal{A}}(f_{\mathcal{S}p}(q), a) \in \mathcal{A}[\mathcal{S}p] \Rightarrow f_{\mathcal{S}p}(\delta^{\mathcal{H}}(q, a)) = \delta^{\mathcal{A}}(f_{\mathcal{S}p}(q), a)$

It is clear that augmenting $\mathcal{S}p$ while maintaining (I1) will eventually lead to all of the (finitely many) states of $\mathcal{A}$ being discovered. If this is the case, $f_{\mathcal{S}p}$ becomes a *bijection*. Combined with (I2) and (I3), $f_{\mathcal{S}p}$ satisfies the requirements of an *isomorphism* (cf. Definition 2).

Relation to Consistency and Closedness. The aforementioned properties should not be confused with the requirements of *consistency* and *closedness*, as introduced in the context of the L* algorithm (Angluin, 1987): while the former are *syntactic* properties referring to the concrete data structure of an observation table, invariants (I1)–(I3) refer to the abstraction of the state space through prefixes that is inherent to many non-observation table-based learning algorithms, too. Also note that closedness and consistency are *preconditions* for constructing a hypothesis from an observation table, while this hypothesis is the *subject* of (I1)–(I3).

The original counterexample handling mechanism in L* actually constitutes a violation of (I1), as multiple short prefixes (upper row labels in the observation table) might represent the same state. The requirement of *consistency* however reconciles this, as it ensures that any subset of representatives for all states can be chosen without the concrete choice affecting the outcome.

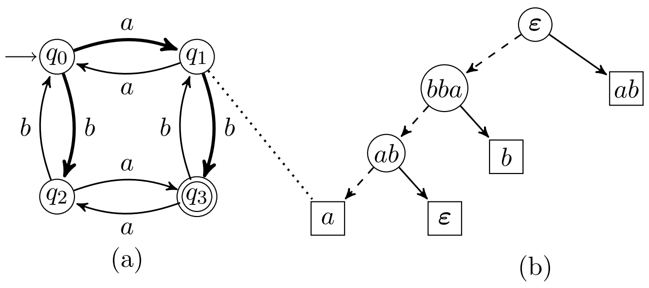

Figure 1: (a) Possible hypothesis DFA $\mathcal{H}$, (b) possible discrimination tree for $\mathcal{H}$.

3.1.1. Discrimination Trees

The main data structure of the Observation Pack algorithm is the *discrimination tree* (Kearns and Vazirani, 1994). The purpose of a discrimination tree is to *organize observations* from past membership queries in an efficient manner that allows maintaining (I1)–(I3). The role of a discrimination tree and its connection to a hypothesis $\mathcal{H}$ are shown in Figure 1. First, note that some of the transitions of the DFA $\mathcal{H}$ in Figure 1a are drawn in bold. These transitions correspond to the access sequences in $\mathcal{S}p = \{\varepsilon, a, ab, b\}$, and they form a *spanning tree* in $\mathcal{H}$.

We will now describe how the discrimination tree shown in Figure 1b corresponds to $\mathcal{H}$. A discrimination tree is a full, rooted binary tree. Leaves are labeled with short prefixes from $\mathcal{S}p$. Hence, there is a one-to-one correspondence between leaves in the discrimination tree and states in $\mathcal{H}$ (dotted line). Inner nodes are labeled with arbitrary words ("discriminators") $v \in \Sigma^*$. Every inner node has two children, the 0-child (dashed line) and the 1-child (solid line). The discrimination tree reflects information gathered from past membership queries as follows: a short prefix $u \in \mathcal{S}p$ labels a leaf in the o-subtree ($o \in \mathbb{B}$) of an inner node labeled by v if $\lambda^{\mathcal{A}}(u \cdot v) = o$ has previously been observed. Therefore, a discrimination tree ensures (I1): for $u \neq u' \in \mathcal{S}p$, the *least common ancestor* of the respective leaves contains a discriminator separating $\mathcal{A}[u]$ and $\mathcal{A}[u']$.

The root of a discrimination tree is always labeled by ε. This allows for deriving the acceptance of a state in $\mathcal{H}$ from the discrimination tree: the state in $\mathcal{H}$ corresponding to $u \in \mathcal{S}p$ is accepting iff u is in the 1-subtree of the root. This implies that $\lambda^{\mathcal{A}}(u \cdot \varepsilon) = 1$ has previously been observed, thus ensuring (I2).

Before looking at (I3), we first need to discuss how transitions in $\mathcal{H}$ are defined. Essential for this is the operation of *sifting* a word $w \in \Sigma^*$ into a discrimination tree: at every inner node labeled by $v \in \Sigma^*$, we branch to either the 0- or the 1-child, depending on $\lambda^{\mathcal{A}}(w \cdot v)$. This procedure is repeated until a leaf is reached, the label of which forms the result of the sifting operation.

Considering a state $q \in Q^{\mathcal{H}}$ represented by its access sequence u, the a-successor, $a \in \Sigma$, of q is determined by sifting ua into the discrimination tree. Let u' be the result of this sifting operation, we then define $\delta^{\mathcal{H}}(q, a) = \mathcal{H}[u']$. If we have any indication that this transition is not correctly wired, i.e., $\delta^{\mathcal{A}}(\mathcal{A}[u], a) \neq \mathcal{A}[u']$, we know for certain that the real a-successor $\mathcal{A}[ua]$ has not been discovered yet, thus preserving (I3): while sifting ua

into the tree, by arriving at u' we have definitely ruled out every other discovered state in $\mathcal{A}[\mathcal{S}p \setminus \{u'\}]$ as a possible a-successor of $\mathcal{A}[u]$.

3.1.2. Counterexample Analysis

If the learning invariants (I1)–(I3) are maintained, augmenting $\mathcal{S}p$ will eventually yield a correct hypothesis $\mathcal{H}$. We have remarked in Section 3 that equivalence queries (or, more precisely, counterexamples) are vital for terminating with a correct result. Therefore, analyzing a counterexample $w \in \Sigma^*$ should result in $\mathcal{S}p$ being augmented.

First, let us remark that we can assume counterexamples to be of positive length, i.e., $w \in \Sigma^+$. This is due to the fact that $\lambda^{\mathcal{A}}(\varepsilon) = \lambda^{\mathcal{H}}(\varepsilon)$, which follows from $\varepsilon \in \mathcal{S}p$ and invariant (I2).

The following result is originally due to Rivest and Schapire (1993), and has been further clarified by Steffen et al. (2011).

Theorem 4 (Counterexample Decomposition) *Let $\mathcal{H}$ be a hypothesis satisfying (I1)–(I3), and let $w \in \Sigma^+$ be a counterexample, i.e., $\lambda^{\mathcal{A}}(w) \neq \lambda^{\mathcal{H}}(w)$. Then, there exists a* decomposition *$\langle u, a, v\rangle \in \Sigma^* \times \Sigma \times \Sigma^*$ such that $w = u \cdot a \cdot v$, and $\lambda^{\mathcal{A}}(\lfloor u \rfloor_{\mathcal{H}}\, a \cdot v) \neq \lambda^{\mathcal{A}}(\lfloor ua \rfloor_{\mathcal{H}} \cdot v)$.*

Let $q = \mathcal{H}[u]$ be the state in $\mathcal{H}$ reached by u. Apparently, $\delta^{\mathcal{H}}(q, a) = q'$ is not the correct target state (wrt. $\mathcal{A}$), as $\lambda^{\mathcal{A}}(\lfloor q' \rfloor_{\mathcal{H}} \cdot v) \neq \lambda^{\mathcal{A}}(\lfloor q \rfloor_{\mathcal{H}}\, a \cdot v)$. Therefore, by invariant (I3) we have $\mathcal{A}[\lfloor u \rfloor_{\mathcal{H}}\, a] \notin \mathcal{A}[\mathcal{S}p]$, meaning we can add $\lfloor u \rfloor_{\mathcal{H}}\, a$ to $\mathcal{S}p$ without violating (I1). This results in an additional state in the hypothesis, and is reflected in the discrimination tree by *splitting* the leaf labeled by $\lfloor ua \rfloor_{\mathcal{H}}$: the leaf is replaced by an inner node labeled by v, and its children are now leaves labeled by $\lfloor ua \rfloor_{\mathcal{H}}$ and $\lfloor u \rfloor_{\mathcal{H}}\, a$.

4. An Abstract Framework for Counterexample Analysis

In this section, we establish our abstract framework for counterexample analysis. We will show that this framework allows us to formulate and implement counterexample analysis algorithms in a very concise and simple manner. This facilitates the understanding of both what the algorithm does, and why it is correct. We will demonstrate these aspects of our framework by using it to formulate the counterexample analysis algorithm due to Rivest and Schapire (1993). Optimizations and alternative approaches will be discussed in Section 5.

4.1. Prefix Transformations

Essential for the above Theorem 4 is the procedure of transforming a *prefix* of a counterexample $w \in \Sigma^+$ into an access sequence in $\mathcal{S}p$. This gives rise to the following definition.

Definition 5 (Prefix Transformation) *Let $\mathcal{H}$ be a hypothesis. The* prefix transformation *with respect to $\mathcal{H}$, $\pi_{\mathcal{H}}$, is defined as follows:*

$$\pi_{\mathcal{H}} \colon \Sigma^* \times \mathbb{N} \to \Sigma^*, \quad \pi_{\mathcal{H}}(w, i) = \left\lfloor w_{[0,i)} \right\rfloor_{\mathcal{H}} \cdot w_{[i,|w|)}.$$

Note that, for $w \in \Sigma^*$, $\pi_{\mathcal{H}}(w, 0) = w$ and $\pi_{\mathcal{H}}(w, |w|) = \lfloor w \rfloor_{\mathcal{H}} \in \mathcal{S}p$. Theorem 4 can thus be rephrased as follows: given a counterexample $w \in \Sigma^+$, there exists an $i \in [0, |w|)$ such that $\lambda^{\mathcal{A}}(\pi_{\mathcal{H}}(w, i)) \neq \lambda^{\mathcal{A}}(\pi_{\mathcal{H}}(w, i+1))$.

In the following, we fix the counterexample $w \in \Sigma^+$, and assume its length to be $|w| = m$. As w is a counterexample wrt. $\mathcal{H}$, the hypothesis $\mathcal{H}$ (wrongly) predicts the output $\lambda^{\mathcal{H}}(w)$ for w. Apparently, prefix transformations have no effect when considering $\lambda^{\mathcal{H}}$: for $0 \leq i \leq m$, $\lambda^{\mathcal{H}}(\pi_{\mathcal{H}}(w,i)) = \lambda^{\mathcal{H}}(w)$.

Since $\pi_{\mathcal{H}}(w,m) \in \mathcal{S}p$, by invariant (I2) we have $\lambda^{\mathcal{A}}(\pi_{\mathcal{H}}(w,m)) = \lambda^{\mathcal{H}}(w)$. Similarly, since $\pi_{\mathcal{H}}(w,0) = w$ and w is a counterexample, $\lambda^{\mathcal{A}}(\pi_{\mathcal{H}}(w,0)) \neq \lambda^{\mathcal{H}}(w)$. We extend this consideration to all indices $0 \leq i \leq m$, compactly representing it in the mapping α defined as follows:

$$\alpha\colon [0, m+1) \to \mathbb{B}, \quad \alpha(i) = \begin{cases} 1 & \text{if } \lambda^{\mathcal{A}}(\pi_{\mathcal{H}}(w,i)) = \lambda^{\mathcal{H}}(w) \\ 0 & \text{otherwise} \end{cases} .$$

The above considerations for indices 0 and m translate to $\alpha(0) = 0$ and $\alpha(m) = 1$. Since $\mathbb{B}$ is two-valued, $\lambda^{\mathcal{A}}(\pi_{\mathcal{H}}(w,i)) \neq \lambda^{\mathcal{A}}(\pi_{\mathcal{H}}(w,i+1))$ is equivalent to $\alpha(i) \neq \alpha(i+1)$. This allows for concisely rephrasing the suffix-based counterexample analysis problem:

Theorem 6 (Counterexample Analysis Rephrased) *Suffix-based counterexample analysis in active automata learning can be rephrased as the problem of, given a mapping $\alpha\colon [0, m+1) \to \mathbb{B}$ with $\alpha(0) = 0$ and $\alpha(m) = 1$, finding an index $\hat{\imath}$, $0 \leq \hat{\imath} < m$, satisfying $\alpha(\hat{\imath}) \neq \alpha(\hat{\imath}+1)$.*

The respective index $\hat{\imath}$ translates to the decomposition $u = w_{[0,\hat{\imath})}$, $a = w_{\hat{\imath}}$, $v = w_{[\hat{\imath}+1,m)}$. It hence yields a suffix of length $m - \hat{\imath}$. The existence of an index $\hat{\imath}$ satisfying $\alpha(\hat{\imath}) \neq \alpha(\hat{\imath}+1)$ (i.e., the existence of a decomposition satisfying the requirements of Theorem 4) is a trivial consequence of $\alpha(0) = 0$ and $\alpha(m) = 1$.[3]

Evaluating $\alpha(i)$, $0 < i < m$, requires a single membership query. The number of membership queries required for analyzing a counterexample is thus equivalent to the number of such evaluations.

A straightforward way to determine such an index $\hat{\imath} \in [0, m)$ satisfying $\alpha(\hat{\imath}) \neq \alpha(\hat{\imath}+1)$ is to scan through α linearly in descending order, and terminating as soon as the value 0 is encountered. Progressing in descending order guarantees finding the largest possible $\hat{\imath}$, resulting in the decomposition with the shortest suffix v. This approach is similar to the Suffix1by1 heuristic due to Irfan et al. (2010) (which however is limited to observation tables in its applicability). While it may in many cases terminate quickly, in the worst-case it requires $m - 1$ membership queries.

4.2. Rivest&Schapire's Method: Binary Search

A linear worst-case query complexity might at first not seem bad. However, in many cases equivalence queries can only be approximated by means of membership queries (e.g., by random sampling). Such approaches yield non-optimal counterexamples (Irfan et al., 2010), which may be significantly longer than the shortest possible ones.

Rivest and Schapire (1993) proposed a counterexample analysis algorithm which drastically reduces the number of membership queries required for finding a suitable decomposition (cf. Theorem 4) to $\lceil \log m \rceil$. This algorithm is based on binary search, and its correctness becomes apparent when formulated in our framework (Algorithm 1). For $0 \leq l < h \leq m$,

3. This can be interpreted as a discrete variant of the well-known *intermediate value theorem* from continuous function analysis.

Algorithm 1 Rivest-Schapire(α)

```
Require: Abstract CE $\alpha\colon [0, m+1) \to \mathbb{B}$
Ensure: Index $\hat{i}$ satisfying $\alpha(\hat{i}) \neq \alpha(\hat{i}+1)$
  $low \leftarrow 0$
  $high \leftarrow m$
  while $high - low > 1$ do
    $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$
    if $A(mid) = 0$ then
      $low \leftarrow mid$
    else
      $high \leftarrow mid$
    end if
  end while
  return $low$
```

Algorithm 2 Find-Exponential(α)

```
Require: Abstract CE $\alpha\colon [0, m+1) \to \mathbb{B}$
Ensure: Index $\hat{i}$ satisfying $\alpha(\hat{i}) \neq \alpha(\hat{i}+1)$
  $ofs \leftarrow 1$,  $high \leftarrow m$,  $low \leftarrow 0$,  $found \leftarrow$ false
  while $high - ofs > 0$ and $\neg found$ do
    if $A(high - ofs) = 0$ then
      $low \leftarrow high - ofs$
      $found \leftarrow$ true
    else
      $high \leftarrow high - ofs$
      $ofs \leftarrow 2 \cdot ofs$
    end if
  end while
  return Binary-Search($\alpha, low, high$)
```

$\alpha(l) = 0$ and $\alpha(h) = 1$ implies that an index $\hat{i}$ satisfying the requirements of Theorem 6 exists in the interval $[l, h)$. Since $\alpha(0) = 0$ and $\alpha(m) = 1$, throughout the execution of Algorithm 1 the invariant $\alpha(low) = 0$ and $\alpha(high) = 1$ is preserved.

Note that Algorithm 1 contains no **break** or **return** statements that could cause premature termination in its main loop. This flavor of binary research is commonly (e.g., in Wirth (1985)) referred to as *deferred detection of equality.* Algorithm 1 thus always requires $\lceil \log m \rceil$ queries.

5. Improved Counterexample Analysis

In this section, we will make use of our abstract framework by presenting optimized variations of Algorithm 1. These algorithms employ different search strategies, which aim for reducing the number of counterexamples, the average length of the suffix (i.e., maximizing $\hat{i}$), or both. Even though the search strategies are well-known (see, e.g., Cormen et al. (2001)), their application in the context of counterexample analysis in active automata learning is novel.

5.1. Exponential Search

Even though we have $\alpha(0) = 0$ and $\alpha(m) = 1$, the value of $\alpha(\cdot)$ may oscillate in between, i.e., α is not necessarily monotonic. An apparent disadvantage of Rivest&Schapire's algorithm is that it tends to result in relatively long counterexamples: if the first *mid* value tested in Algorithm 1 happens to satisfy $\alpha(mid) = 1$, the resulting suffix will be at least of length $\lceil m/2 \rceil$. A way of favoring short suffixes while maintaining the logarithmic worst-case query complexity of Rivest&Schapire's algorithm is to apply *exponential search* (in descending order). This means testing $\alpha(m - 2^0)$, $\alpha(m - 2^1)$, $\alpha(m - 2^2)$ etc., until a range $[l, h)$ satisfying $\alpha(l) = 0, \alpha(h) = 1$ emerges, which is then searched for an index $\hat{i}$ using binary search (cf. Alg. 1). Algorithm 2 shows the algorithm. Again, the key invariant we maintain is $\alpha(low) = 0, \alpha(high) = 1$.

In the worst-case (if *found* never becomes true), Algorithm 2 requires $2\lfloor \log m \rfloor$ membership queries: $\lfloor \log m \rfloor$ for the (unsuccessful) exponential search, and another $\lfloor \log m \rfloor$ for the binary search on the left half of α. In practice, the search may terminate much earlier, requiring only a single membership query in the best case (if $\alpha(m-1) = 0$).

Algorithm 3 Partition-Search(α)

```
Require: Abstract CE $\alpha\colon [0, m+1) \to \mathbb{B}$
Ensure: Index $\hat{i}$ satisfying $\alpha(\hat{i}) \neq \alpha(\hat{i}+1)$
  $step \leftarrow \lfloor \frac{m}{\log m} \rfloor$,  $low \leftarrow 0$,  $high \leftarrow m$
  $found \leftarrow$ false
  while $high - step > low$ and $\neg found$ do
    if $\alpha(high - step) = 0$ then
      $low \leftarrow high - step$
      $found \leftarrow$ true
      break
    else
      $high \leftarrow high - step$
    end if
  end while
  return Binary-Search($\alpha, low, high$)
```

Algorithm 4 RS-Eager(β)

```
Require: Mapping $\beta\colon [0, m) \to \{0, 1, 2\}$
Ensure: Index $\hat{i}$ satisfying $\beta(\hat{i}) = 1$
  $low \leftarrow 0$
  $high \leftarrow m - 1$
  while $high > low$ do
    $mid \leftarrow \left\lfloor \frac{low+high}{2} \right\rfloor$
    if $\beta(mid) = 1$ then
      return $mid$
    else if $\beta(mid) = 0$ then
      $low \leftarrow mid + 1$  // $\beta(mid+1) \leq 1$
    else
      $high \leftarrow mid - 1$  // $\beta(mid-1) \geq 1$
    end if
  end while
  return $low$
```

5.2. Partition Search

Exponential search may terminate quickly and with very short suffixes. However, the exponentially fast growth of the search range may be disadvantageous in the unfortunate event that the first few positions of form $m - 2^i$ that are tested all satisfy $\alpha(m - 2^i) = 1$.

A more balanced approach that still achieves a logarithmic worst-case query complexity is to partition α into $\lceil \log m \rceil$ fields, each of approximate length $s = \lfloor \frac{m}{\log m} \rfloor$. Then, the values $\alpha(m-s)$, $\alpha(m-2s)$ etc. are tested, until a range $[l, h)$ satisfying $\alpha(l) = 0, \alpha(h) = 1$ emerges. Again, binary search is used to find $\hat{i}$ in this range, requiring an additional $\lceil \log s \rceil = \lceil \log \lfloor \frac{m}{\log m} \rfloor \rceil$ membership queries.

This procedure is detailed in Algorithm 3. The invariant $\alpha(low) = 0, \alpha(high) = 1$ is preserved throughout the algorithm. In the worst case, $\lceil \log m \rceil + \lceil \log \lfloor \frac{m}{\log m} \rfloor \rceil = O(\log m)$ membership queries are necessary. Note that the first term is an upper bound (searching for a suitable range), while the second term is unavoidable (binary search). Thus, as m grows larger, the proportion of the unavoidable term grows larger as well. We therefore expect this algorithm to perform better for shorter counterexamples.

5.3. Eager Search

The binary search in Algorithm 1 always requires at least $\lfloor \log m \rfloor$ iterations, as the **while**-loop contains no **break** or **return** statements. In the classical application of binary search, this is referred to as *deferred detection of equality* (Wirth, 1985). This realization is due to the fact that the mere value of $\alpha(i)$ for some $i \in [0, m)$ is never sufficient to determine if i satisfies the requirements from Theorem 6, as this requires to test $\alpha(i+1)$ as well.

However, we can aggregate the values of neighboring elements as follows. Consider the mapping β, derived from $\alpha\colon [0, m+1) \to \mathbb{B}$:

$$\beta\colon [0, m) \to \{0, 1, 2\}, \quad \beta(i) = \alpha(i) + \alpha(i+1).$$

Apparently, $\alpha(i) \neq \alpha(i+1)$ iff $\beta(i) = 1$. While evaluating $\beta(i)$ may require up to 2 membership queries, any search may terminate immediately if a value of 1 has been encountered.

We illustrate the effect of eager search on Rivest&Schapire's algorithm only (cf. Algorithm 1), even though it may be applied to virtually any other abstract counterexample analysis algorithm as well. The resulting algorithm is shown in Algorithm 4. Note that, since $\alpha(0) = 0$ and $\alpha(m) = 1$, we have $\beta(0) \leq 1$ and $\beta(m-1) \geq 1$. Also, consecutive values of $\beta(\cdot)$ differ by at most 1. This makes clear why Algorithm 1 works: throughout the algorithm, we maintain the invariant that $\beta(\mathit{low}) \leq 1$ and $\beta(\mathit{high}) \geq 1$.[4]

6. Application to Kearns&Vazirani's Algorithm

Before moving on to the evaluation, we want to highlight the generality of our framework by showing its applicability to a different active automata learning algorithm: the one by Kearns and Vazirani (1994). This algorithm, which was the first one to use a discrimination tree in the context of active learning, can be seen as *dual* to the Observation Pack algorithm, reversing the way prefixes and suffixes are treated.

The Observation Pack algorithm maintains a *prefix-closed* set of *short prefixes* identifying states in the hypothesis. Due to prefix-closedness and the fact that there is only one short prefix per state, the length of short prefixes is bounded by n. Discriminators—which label inner nodes in the discrimination tree—, are extracted as *suffixes* of former counterexamples, and follow no particular pattern. The length of discriminators is only bounded by m, where m is the length of the longest counterexample.

Looking at the Kearns&Vazirani algorithm, we are facing almost the exact opposite: the set of the state-identifying short prefixes is no longer prefix-closed, but they are extracted as *prefixes* of former counterexamples and thus are only bounded by m. The separation of states in the hypothesis is again maintained in a *discrimination tree*, but this time the set of discriminators forms a *suffix-closed* set: every new discriminator that is added to the tree is formed by prepending a single symbol to an existing discriminator. This ensures that the length of discriminators is bounded by n.

6.1. Counterexample Analysis

The counterexample analysis in Kearns&Vazirani's algorithm is rather straightforward: given a hypothesis $\mathcal{H}$ and a counterexample w of length m, prefixes of w are considered in ascending order of their length. The prefix $w_{[0,i)}$ is sifted into the discrimination tree, until the resulting leaf differs from the leaf associated with $\mathcal{H}[w_{[0,i)}]$. Let then v be the label of the *lowest common ancestor* (LCA) of these two leaves in the discrimination tree. By prepending $a = w_{i-1}$ to v, we obtain a new discriminator which separates $\lfloor w_{[0,i-1)} \rfloor_{\mathcal{H}}$ from $w_{[0,i-1)}$. The leaf corresponding to $\mathcal{H}[w_{[0,i-1)}]$ is then split, using av as the discriminator and $w_{[0,i-1)}$ as the representative for the new state.

From this perspective, it seems impossible to apply Rivest&Schapire's counterexample analysis algorithm, as there is no natural notion of *prefix transformations*. Furthermore, suffixes that are added to the discrimination tree have to be incrementally constructed from existing suffixes. Consequently, there presently do not exist—to the best of our knowledge—any approaches to replace the linear scanning from the original algorithm with something more efficient (i.e., of sublinear worst-case complexity). However, in the following we show how the counterexample analysis problem for Kearns&Vazirani's algorithm can

4. Again in analogy to the intermediate number theorem, we can conclude the existence of $\hat{\imath}$, $\mathit{low} \leq \hat{\imath} \leq \mathit{high}$, satisfying $\beta(\hat{\imath}) = 1$.

be abstracted in a way which makes it amenable to the techniques developed in this paper. Note that the main challenge here is again to transform the *concrete* counterexample w into an *abstract* counterexample α. Once this is accomplished, the algorithms discussed in the previous sections can be applied without further adaptations.

6.2. Counterexample Abstraction

Looking at the counterexample analysis procedure as described above, it is clear that we are faced with the same abstract problem as in the Observation Pack algorithm: to find an index i in the counterexample w which satisfies a given property (here: that the leaf corresponding to $\mathcal{H}[w_{[0,i)}]$ matches the result of the respective sift operation), while $i+1$ does not. An *abstract counterexample* α in the context of the Kearns&Vazirani algorithm can thus be derived as follows:

$$\alpha\colon [0, m+1) \to \mathbb{B}, \quad \alpha(i) = \begin{cases} 0 & \text{if } \mathcal{H}[w_{[0,i)}] = \mathit{sift}(w_{[0,i)}) \\ 1 & \text{otherwise} \end{cases} .$$

Note that again we have $\alpha(0) = 0$ (the empty word is identified with the initial state) and $\alpha(m) = 1$ (the counterexample is classified incorrectly, hence $\mathcal{H}[w_{[0,i)}]$ and $\mathit{sift}(w_{[0,i)})$ must be in different subtrees of the root node).

This is already sufficient for enabling applicability of the counterexample analysis framework presented in the paper. There is, however, a slight difference in practice: in the context of the Observation Pack algorithm, it is desirable to minimize the length of the extracted discriminators. Therefore, it makes sense to start searching (e.g., in linear scanning or using exponential search) from the *end* of the counterexample. In the context of the Kearns&Vazirani algorithm, however, for minimizing the length of the extracted *prefixes* one needs to start from the *beginning* of the counterexample.

7. Implementation and Evaluation

We have implemented our abstract framework, as well as the algorithms we presented, on top of *LearnLib*. LearnLib[5] is our open-source framework for active automata learning, and is a reimplementation of the former closed-source version of LearnLib (Merten et al., 2011). We have made our implementation, along with the environment for running experiments (including scripts etc.), publicly available.[6] We plan to fully integrate our work into LearnLib in the near future.

We conducted a small evaluation to illustrate the impact of the optimizations presented in Section 5 for both the Observation Pack and Kearns&Vazirani's (cf. Sec. 6) algorithm. For a randomly generated DFA with 500 states and an alphabet of size 10,[7] we measured the total number of membership queries during all counterexample analyses (Fig. 2, top) and the average length of the returned suffixes/prefixes (Fig. 2, bottom). We did not consider

5. `http://www.learnlib.de/`
6. `https://github.com/misberner/learnlib-abstract-counterexamples`
7. Both changes in the size parameters and different outcomes of random DFA generator did not have a major impact on the relative performance and the shape of the plots. We therefore consider it sufficient to focus on a single instance of a randomly generated DFA only, especially since the evaluation resources are publicly available. Whether randomly generated DFA are "representative" for typical automata learning setups is a different question, which however is beyond the scope of this paper.

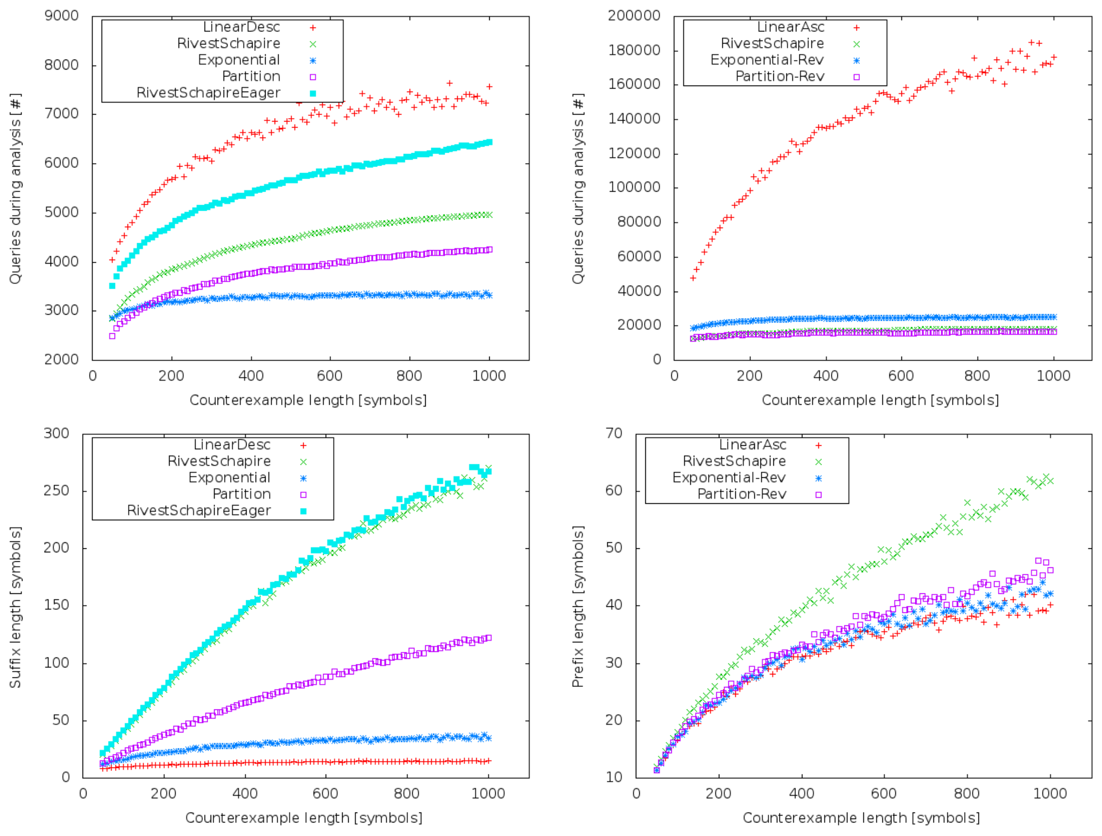

Figure 2: Average total number of membership queries during counterexample analysis (top) and average length of returned suffix/prefix (bottom), plotted against counterexample length for both Observation Pack (left) and Kearns&Vazirani (right) algorithms

the number of membership queries not related to counterexample analysis, as we found it to be by and large independent of the search strategy. We used the Observation Pack algorithm (called *Discrimination Tree* algorithm in LearnLib), since it guarantees that the number of counterexample analyses performed depends solely on the number of states of the target automaton; the same is true for Kearns&Vazirani's algorithm. We measured these numbers as functions of the counterexample length, which we fixed for each learning process. Counterexamples were generated by randomly sampling words of the required length, and considering only those words that were in fact counterexamples. Finally, each experiment was run 50 times in an effort to average out variations due to random counterexample sampling.

Observation Pack. The results are shown in the left column of Figure 2 in graphical form. As remarked in Section 4.1, linear scanning through the counterexample in descending order yields the shortest suffixes, but is the most expensive approach in terms of membership queries. Our results also show that an eager version of Rivest&Schapire's algorithm neither helps reducing the cost in terms of membership queries (in fact, the increase is quite significant), nor does it yield shorter suffixes as the original version.

Partition search and exponential search outperform Rivest&Schapire's algorithm in both measures. As predicted, partition search requires slightly less membership queries for relatively short counterexamples (up to 100 symbols), but then is quickly outperformed by exponential search. Furthermore, the latter yields suffixes of length much closer to the relative optimum (as established by `LinearDesc`) than any other algorithm.

Kearns&Vazirani. Our observations for Kearns&Vazirani's algorithm are slightly different: the performance of linear scanning (which is the approach originally described by Kearns and Vazirani (1994)), measured in terms of the number of queries, is extremely poor for long counterexamples.[8] The other search strategies perform much better. Interestingly, both Rivest&Schapire (i.e., binary search) and partition search outperform exponential search. However, in spite of that, exponential search yields shorter prefixes than the former two, only being outperformed by linear scanning. This makes exponential search a good all-rounder in this context, even though its advantages are not as clear as when employed for counterexample analysis in the context of the Observation Pack algorithm.

8. Conclusion

We have presented an abstract framework for counterexample analysis in active automata learning. This framework enables a perspective on counterexample analysis focusing on the essentials: the problem of finding an adequate suffix index is reduced to finding distinct neighbors in an array with distinct first and last elements. We have shown that this framework facilitates formulating algorithms concisely, and with evident correctness properties.

An instantiation of the framework, based on *prefix transformations*, was used to specify variations to the counterexample analysis algorithm due to Rivest and Schapire (1993) that, while maintaining the worst-case upper bound of $O(\log m)$ membership queries, require much less queries in practice and yield shorter suffixes. A second instantiation of the framework was demonstrated for the algorithm due to Kearns and Vazirani (1994), demonstrating the framework's flexibility: the same (abstract) analysis procedures can be used for both algorithms, in spite of them posing almost complementary requirements for counterexample analysis. The *sublinear* analysis procedures which our framework enables are—to the best of our knowledge—the first for Kearns&Vazirani's algorithm.

We have implemented our framework and the presented algorithms on top of *LearnLib*, and conducted experiments confirming our hypothesis that these algorithms in practice frequently outperform Rivest&Schapire's algorithm, and the classical, linear scanning-based approach in Kearns&Vazirani's algorithm, respectively.

As our experiments suggest that some of the analysis algorithm we presented yield much shorter suffixes than Rivest&Schapire's algorithm, it will be interesting to investigate how this affects the performance of the recently presented TTT algorithm (Isberner et al., 2014), which aims at completely eliminating any redundancies in the suffix. Finally, it remains an open problem to generalize the established perspective on counterexample analysis to learning algorithms involving several layers of abstraction (e.g., an additional abstraction on the alphabet, as opposed to an abstraction of the state-space only), such as *Alphabet Abstraction Refinement* (Isberner et al., 2013) or *register automata* learning (Howar et al., 2012).

8. Note that counterexample analysis is generally more expensive as in the Observation Pack algorithm, as evaluating $\alpha(i)$ requires *sifting* of the respective prefix.

If it was possible to extend the notion of abstract counterexamples to these settings, all the algorithms we discussed would be usable at once, without any additional implementation effort.

References

Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

Falk Howar. *Active Learning of Interface Programs*. PhD thesis, TU Dortmund University, 2012. URL http://dx.doi.org/2003/29486.

Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring Canonical Register Automata. In *VMCAI*, volume 7148 of *LNCS*, pages 251–266. Springer, 2012.

Muhammad Naeem Irfan. *Analysis and Optimization of Software Model Inference Algorithms*. PhD thesis, Université de Grenoble, France, September 2012.

Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz. Angluin style finite state machine inference with non-optimal counterexamples. In *Proceedings of the First International Workshop on Model Inference In Testing*, 2010.

Malte Isberner, Falk Howar, and Bernhard Steffen. Inferring Automata with State-local Alphabet Abstractions. In *NFM*, volume 7871 of *LNCS*, pages 124–138, 2013.

Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In *Runtime Verification*. Springer-Verlag Berlin Heidelberg, 2014. to appear.

Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In *TACAS*, pages 220–223, 2011.

A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. ISSN 00029939.

Ronald L. Rivest and Robert E. Schapire. Inference of Finite Automata Using Homing Sequences. *Inf. Comput.*, 103(2):299–347, 1993. ISSN 0890-5401.

Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to Active Automata Learning from a Practical Perspective. In *SFM*, volume 6659 of *LNCS*, pages 256–296. Springer, 2011.

Niklaus Wirth. *Programming in MODULA-2 (3rd Corrected Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1985. ISBN 0-387-15078-1.