Master Thesis
Computing Science

Radboud University

Enhancing Active Automata Learning by a User Log Based Metric

Author
Petra van den Bos

First Supervisor
prof. dr. Frits W. Vaandrager

Second Supervisor
Rick Smetsers

Nijmegen
July 6, 2015

Contents

Abstract

Active automata learning enables to learn a model from a real world system. Such a model is constructed iteratively by adapting intermediate models, until no counterexample can be found that distinguishes the intermediate model and the real world system. It is often not clear however, that an intermediate model is better than its predecessor. This thesis presents a user log based metric to measure distances between two models. A user log is used to reach some 'sensible' state in each of the models. From those states, a shortest counterexample is found, that distinguishes the two models. The length of this counterexample determines the distance: the longer it is, the smaller the distance. Hence, an intermediate model is better than its predecessor, if it is closer to the final model.

Finding counterexamples of minimal length is not trivial. Therefore, algorithms for finding them are presented. Furthermore, it is shown that the user log metric can be embedded in the standard active automata learning algorithm, without having to know the final model beforehand. In this adapted algorithm, an intermediate model is never more distant from the final model than its predecessor. On top of that, first experiments yielded a 24% reduction of the input symbols that have to be send to the real world system, during the execution of the adapted learning algorithm.
Keywords: Active automata learning, distance metric, user log, minimal length counterexample

1 Introduction

Real world software systems are often complex. Additionally, they are frequently adapted to keep up with a changing environment. This makes it hard to know exactly how a system behaves. There are however ways to help with this problem. One possibility is to learn a model of the system automatically (De la Higuera, 2010), for instance a finite automaton representation. It gives a more abstract overview of the system, since it only shows what behaviour the system has, and not how the system realizes this behaviour.

In this thesis, automata learning is used as a black box technique. Only observations from the outside are used to learn a model. These observations consist of the inputs that can be given to the system, and the outputs the system can return as a response to some input. Black box methods are suitable if the system has components written in several languages, if it has hardware components, or if the code base is too big or too hard to understand. White box methods are preferable whenever it is possible to use them, since the internal structure of the system helps in analysing it.

There exist two variants of learning: passive automata learning and active automata learning. In passive learning (Murphy, 1995; Freund et al., 1997), a model is build from a large collection of user logs from the system. It is assumed that the collection of logs captures all the behaviour of the system.

In active learning (Angluin, 1987; Steffen et al., 2011), a model is established by communicating directly with the system itself. This model is learned iteratively. First an initial model is constructed based on the outputs the system produces for some set of inputs. After that, the model is adapted repeatedly until no counterexample can be found any more. A counterexample is a sequence of inputs for which the current model produces a different sequence of outputs than the system.

In practice it is often not evident when an intermediate model is better than the previous intermediate model. The counterexample used in the adaptation probably has no obvious semantics, as the learning algorithm has no knowledge about the meaning and purpose of the system. The algorithm simply takes the first counterexample it can find.

A metric would help to establish whether intermediate models get better each time. With a metric the distance between the current model and final model of the system can be calculated. If a model is closer to the final model than its predecessor, it is better than its predecessor.

Smetsers et al. (2014) proposed such a metric. They used the length of the counterexamples, i.e. the number of input symbols in the sequence. A short counterexample is more general than a long counterexample. A counterexample still stays a counterexample if input symbols are appended to it. However, a prefix of a counterexample is not always a counterexample. This is the case when the system produces the same outputs for the inputs of the prefix, but different outputs for inputs after the prefix. By demanding that counterexamples are always of the shortest possible length, the length determines the distance between two models. The longer a counterexample, the closer it is to the final model.

The metric uses the original counterexamples found by the active learning algorithm.

As explained earlier, those counterexamples likely do not have any obvious semantics. Moreover, all counterexamples are counterexamples for the initial state of the intermediate model. Other states may be of interest as well, since they may have a meaning and some states are probably visited much more often in a typical usage scenario of the system, than the initial state. Hence, it makes sense to first go to a 'sensible' state, and then search for a counterexample starting in that state. The length of that counterexample can then be used in the same way to define the metric as the metric of Smetsers et al. (2014).

User logs provide a way to find 'sensible' states. A user log is an input sequence that is provided to the system by some user. A user can be an actual person, or another system that communicates with it. A user will often want to achieve some goal by providing those inputs. Thus it is safe to assume that a user log has some meaning, although it may occasionally happen that a user makes a mistake and provides the wrong inputs. In any case, using counterexamples from states reached by user logs is less arbitrary than only using counterexamples starting in the initial state. Any number of user logs can help, but of course, a larger number will probably result in reaching more (meaningful) states.

In practice, it is hard to find a counterexample that has the shortest possible length. By trying excessively many input sequences, a counterexample can be found that approaches the shortest length. For the user log metric to work, an approximation is not enough, since it needs the exact length of a shortest distinguishing string.

There is however an escape: the user log metric not only enables to calculate distances between an intermediate and the final model, but also between two intermediate models that are already found. This will give enough information to decide whether an intermediate model is at least as close to the final model as its predecessor.

Subsequently, the user log metric is defined formally for a general kind of automata. This way, the theory presented in this thesis can be applied on several automata, such as Mealy machines, Moore machines and Deterministic Finite Automata. After that, it is discussed how the user log metric can be embedded in the active learning algorithm, such that intermediate models never get worse than the previous one. Next, algorithms are presented to find a shortest counterexample for two intermediate models. It is proven that those algorithms work correctly and have worst case complexity $\mathbb{O}(n^2)$. Here n is the total number of states of the models the user log metric is applied on. Furthermore, it is shown that the presented theory can be applied in a real world case study. First experiments show that the number of input symbols, needed to learn the final model, is reduced by 24%, when using the adapted active learning algorithm instead of the standard learning algorithm.

2 Applying a metric to active automata learning algorithms

2.1 Models

Active automata learning algorithms can be applied to systems to retrieve a model of the behaviour of those systems. Definition 1 defines an automaton. Throughout this thesis, it will be used to represent models. This way, the same definitions, proofs, and algorithms do not have to be stated multiple times for similar automata. Definition 2 shows that Deterministic Finite Automata, Mealy machines, and Moore machines are instances of automata.

Definition 1 An *automaton* is a 6-tuple $(Q, \Sigma, q_0, D, \delta, obs)$, where

- Q is a finite set of states,
- Σ is a set of input symbols,
- $q_0 \in Q$ is the initial state,
- D is a set of outputs (elements may consist of zero, one or more output symbols),
- $\delta : Q \times \Sigma \to Q$ is a transition function, that establishes the next state for a given state and input symbol, and
- $obs : \Sigma^* \to D$ is an observation function that produces an output, given an input sequence.

The transition function δ is extended in the usual way for sequences of input to $\delta' : Q \times \Sigma^* \to Q$, i.e. $\delta'(s, \upsilon\tau) = \delta'(\delta(\upsilon, s), \tau)$ for $\upsilon \in \Sigma$ and $\tau \in \Sigma^*$, and $\delta'(s, \epsilon) = s$.

For convenience, the observation function of some automaton A is denoted as obs_A, if the elements of A are not mentioned explicitly.

Definition 2 Three special cases of an automaton are given below. The elements Σ, Q, q_0, and δ are defined as in Definition 1.

- A *Deterministic Finite Automata* (DFA) is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$. The set $F \subseteq Q$ contains the accepting states. A DFA is an automaton: define $D = \{\text{accept, reject}\}$ and
  $obs(\tau) = \begin{cases} \text{accept} & \text{if } \delta(q_0, \tau) \in F \\ \text{reject} & \text{if } \delta(q_0, \tau) \notin F \end{cases}$
- A *Mealy machine* is a 6-tuple $(\Sigma, \Gamma, Q, q_0, \delta, \lambda)$. The set Γ is the output alphabet. For each transition that is taken by means of δ, the function $\lambda : Q \times \Sigma \to \Gamma$ produces an output symbol. It is extended in the usual way to $\lambda' : Q \times \Sigma^* \to \Gamma^*$, i.e. $\lambda'(s, \upsilon\tau) = \lambda(s, \upsilon)\lambda'(\delta(s, \upsilon), \tau)$ for $\upsilon \in \Sigma$ and $\tau \in \Sigma^*$, and $\lambda'(s, \epsilon) = \epsilon$. A Mealy machine is an automaton: define $D = \Gamma^*$ and $obs(\tau) = \lambda'(q_0, \tau)$

- A *Moore machine* is a 6-tuple $(\Sigma, \Gamma, Q, q_0, \delta, \omega)$. The set Γ is the output alphabet. The function $\omega : Q \to \Gamma$ produces an output symbol for each state. Define the extension $\omega' : Q \times \Sigma^* \to \Gamma^*$ as $\omega'(s, \upsilon\tau) = \omega(s)\omega'(\delta(s, \upsilon), \tau)$ for $\upsilon \in \Sigma$ and $\tau \in \Sigma^*$, and $\omega'(s, \epsilon) = \omega(s)$. A Moore machine is an automaton: define $D = \Gamma^*$ and $obs(\tau) = \omega'(q_0, \tau)$.

Throughout this thesis, the automaton $L = (Q_L, \Sigma, q_{0_L}, D, \delta_L, obs_L)$ is fixed to be the model of the system, as found by the learning algorithm. The set of often used input sequences is denoted with I. It is a finite subset of Σ^*. The empty input ϵ is always an element of I, as this is the most minimal usage profile a system can have.

Definition 3 defines some basic notions of distinguishing strings and equivalence for automata.

Definition 3 Let A and A' be two automata.

- A string $\rho \in \Sigma^*$ *distinguishes* A and A' if and only if $obs_A(\rho) \neq obs_{A'}(\rho)$.
- Let $\sigma \in I$ be a string. A string $\rho \in \Sigma^*$ distinguishes A and A' *after* σ if and only if $\sigma\rho$ distinguishes A and A'.
- Let $\rho \in \Sigma^*$ be a distinguishing string after some $\sigma \in I$. Then ρ is *minimal* if and only if there exists no other string ρ' that distinguishes A and A' after some $\sigma' \in I$, such that $|\rho'| < |\rho|$. If it is not relevant
- Automaton A and A' are equivalent, denoted with $A \approx A'$, if and only if for all $\tau \in \Sigma^*$ it holds that $obs_A(\tau) = obs_{A'}(\tau)$.
- A string τ is an I-minimal distinguishing string for A and A' if and only if there exist strings $\rho \in \Sigma^*$ and $\sigma \in I$ such that $\tau = \sigma\rho$, and such that ρ is a minimal string that distinguishes A and A' *after* σ.

2.2 Measuring distance between hypotheses

An active automata learning algorithm starts with constructing an initial model. After that, it does equivalence queries to test whether it matches the system. If a counterexample is found, it is used to construct a new model from the current one. This is done again and again, until the final model L is found. The models constructed while learning, are called hypotheses.

Definition 4 is more specific than Definition 3. It defines distinguishing strings and equivalence for two states of two hypotheses. This is used later to construct I-minimal distinguishing strings.

Definition 4 Let $H = (Q, \Sigma, q_0, D, \delta, obs)$ and $H' = (Q', \Sigma, q_0', D', \delta', obs')$ be two hypotheses of an active learning algorithm. Let $s \in Q$ and $s' \in Q'$ be states.

- Since H and H' were constructed by an active automata learning algorithm, s and s' were reached from the initial state for some input strings. Let $\tau, \tau' \in \Sigma^*$ be those accessing strings, i.e. $\delta(q_0, \tau) = s$ and $\delta'(q_0', \tau') = s'$. Then a string $\rho \in \Sigma^*$ *distinguishes* s and s' if and only if $obs(\tau\rho) \neq obs(\tau'\rho)$.

- The states s and s' are equivalent if and only if there exists no string that distinguishes s and s'.

The length of I-minimal distinguishing strings enables to define a metric on hypotheses. As such, a concrete distance between two hypotheses can be calculated. The metric can only be applied on automata that comply with the user profiles, since otherwise it would not be useful to calculate the distance using them. Definition 5 makes this set of allowed automata $\mathcal{A}$ explicit.

Definition 5 Let A be an automaton. Then $A \in \mathcal{A}$ if and only if for all $\sigma \in I$ it holds that $obs_A(\sigma) = obs_L(\sigma)$.

The metric itself is defined in Definition 6. Lemma 7 asserts that it is indeed an (ultra)metric in context of $\mathcal{A}$.

Definition 6 Let $A, A' \in \mathcal{A}$ be two automata. The metric $d : \mathcal{A} \times \mathcal{A} \to \mathbb{Q}$ is defined as:

$$d(A, A') = \begin{cases} 0 & \text{if } A \approx A' \\ 2^{-n} & \text{otherwise} \end{cases}$$

where n is the length of a *minimal* string ρ that distinguishes A and A' *after* some $\sigma \in I$.

Lemma 7 $\langle \mathcal{A}, d \rangle$ is an ultrametric space, i.e.,
For any $A_1, A_2, A_3 \in \mathcal{A}$:

1. $d(A_1, A_2) = 0 \iff A_1 \approx A_2$
2. $d(A_1, A_2) = d(A_2, A_1)$
3. $d(A_1, A_2) \leq \max(d(A_1, A_3), d(A_3, A_2))$

Proof:

1. $\Leftarrow$) By definition of d, this is the case.

   $\Rightarrow$) If it would be the case that $A_1 \not\approx A_2$, there must be some minimal string ρ that distinguishes A_1 and A_2 after some $\sigma \in I$. Then ρ has some length $n \geq 0$, but 2^{-n} is never zero. Hence, $d(A_1, A_2) = 0$.

2. If $A_1 \approx A_2$, this holds trivially, so suppose $A_1 \not\approx A_2$. Let ρ be a minimal string that distinguishes A_1 and A_2 after some $\sigma \in I$. Then $\sigma\rho$ also distinguishes A_2 and A_1, because $\approx$ is a symmetric relation.

3. The proof of this statement is done by applying case distinction:

   (a) If $A_1 \approx A_2$, $d(A_1, A_2) = 0$, which is the lowest value d can output, according to Definition 6. This means that $d(A_1, A_2)$ is smaller or equal to the distance between any two languages, in particular $\max(d(A_1, A_3), d(A_3, A_2))$.

   (b) If $A_1 \approx A_3$, the statements is that $d(A_3, A_2) \leq \max(d(A_3, A_3), d(A_3, A_2)) = \max(0, d(A_3, A_2)) = d(A_3, A_2)$, which holds trivially. The same argument applies to $A_2 \approx A_3$.

   (c) Suppose A_1, A_2 and A_3 are all inequivalent. Let ρ_1 be a minimal string that distinguishes A_1 and A_2 after some $\sigma_1 \in I$, ρ_2 a minimal string that distinguishes A_2 and A_3 after some $\sigma_2 \in I$, and ρ_3 a minimal string that distinguishes A_1 and A_3 after some $\sigma_3 \in I$.
   To prove: $2^{-|\rho_1|} \leq \max(2^{-|\rho_2|}, 2^{-|\rho_3|})$, or equivalently: $|\rho_1| \geq \min(|\rho_2|, |\rho_3|)$. Suppose that the opposite holds: $|\rho_1| < \min(|\rho_2|, |\rho_3|)$. Because ρ_2 and ρ_3 are minimal strings, it holds that for any τ such that $|\tau| < |\rho_2|$ and any $\sigma \in I$, that $obs_{A_2}(\sigma\tau) = obs_{A_3}(\sigma\tau)$. The same way, it holds that for any υ such that $|\upsilon| < |\rho_3|$ and any $\sigma' \in I$, that $obs_{A_3}(\sigma'\upsilon) = obs_{A_1}(\sigma'\upsilon)$. In particular, for $\tau = \upsilon = \rho_1$ and $\sigma = \sigma' = \sigma_1$. But then $\sigma_1\rho_1$ does not distinguish A_1 and A_2, which is a contradiction. $\square$

2.3 An adapted active automata learning algorithm

After running an active automata learning algorithm, the metric d enables to calculate the distance between some intermediate model and the final model. However, it can also be used to enhance the learning process while running. Lemma 9 gives rise to a way to do this. The proof uses that two distances among automata A_1, A_2 and A_3 are equal and the third one is equal or smaller, as stated in Lemma 8.

Lemma 8 Let $A_1, A_2, A_3 \in \mathcal{A}$. Then it holds that:
$d(A_1, A_2) \neq d(A_2, A_3) \Rightarrow d(A_1, A_3) = \max(d(A_1, A_2), d(A_2, A_3))$

Proof: Smetsers et al. (2014) prove this lemma for any ultrametric space $\langle X, d' \rangle$. It applies to $\langle \mathcal{A}, d \rangle$, as Lemma 7 proves that $\langle \mathcal{A}, d \rangle$ is an ultrametric space.

Lemma 9 Let $H, H' \in \mathcal{A}$ be hypotheses with $H \neq H'$. Let ρ_1 be a minimal string that distinguishes H and H' after some $\sigma_1 \in I$. Then it holds that:
$d(L, H) < d(L, H') \Rightarrow \forall \sigma \in I : obs_H(\sigma\rho_1) = obs_L(\sigma\rho_1)$

Proof: Assume $d(L, H) < d(L, H')$. If $d(L, H') = 0$, the lemma holds trivially, so assume $d(L, H') \neq 0$. Let ρ_2 be a minimal string that distinguishes L and H after some $\sigma_2 \in I$, and ρ_3 a minimal string that distinguishes L and H' after some $\sigma_3 \in I$. By Definition 6, $d(L, H) < d(L, H')$ implies that $|\rho_2| > |\rho_3|$. According to Lemma 8, $d(L, H) < d(L, H')$ also implies that $d(L, H') = d(H, H')$, thus $|\rho_1| = |\rho_3|$. Henceforth, $|\rho_2| > |\rho_3|$ and $|\rho_1| = |\rho_3|$

imply that $|\rho_2| > |\rho_1|$. From this it follows that, for any $\sigma \in I$, $\sigma\rho_1$ cannot distinguish L and H. Therefore, L and H must produce the same output for input $\sigma\rho_1$. $\square$

Let H and H' be subsequent hypotheses in the execution of an active automata learning algorithm, and let τ be an I-minimal string that distinguishes H and H'. If $obs_H(\tau) \neq obs_L(\tau)$, then Lemma 9 tells that $d(L, H) \geq d(L, H')$, so H' is indeed closer to L than H. Hence, no action has to be taken.

However, if $obs_H(\tau) = obs_L(\tau)$, it is not certain whether H' is closer to L than H. Nevertheless, $obs_{H'}(\tau) \neq obs_L(\tau)$ is the case, as τ distinguishes H and H'. Thus τ can be used to construct an improved model. This adaptation to an active automata learning algorithm is described in pseudo code by Algorithm 1.

Algorithm 1: Modified active automata learning algorithm

```
   Input: A system under test L that can be reset and return outputs for inputs
 1 Construct initial hypothesis H;
 2 Ask equivalence query for H, let r be the response;
 3 while r contains a counterexample τ do
 4     repeat
 5         Construct H' from handling counterexample τ for H;
 6         for σ ∈ I do
 7             if σ distinguishes H' and L then
 8                 H := H';
 9                 Construct a new H' from handling counterexample σ for H';
10             end
11         end
12         Obtain an I-minimal string τ that distinguishes H and H';
13     until τ distinguishes H and L;
14     H := H';
15     Ask equivalence query for H, let r be the response;
16 end
   Result: The final model H ≈ L
```

It can happen that a hypothesis H is not consistent with I, i.e. there exists some $\sigma \in I$ that distinguishes H and L. The counterexample can then be used to construct a hypothesis that is consistent with L for this string σ. This is done at lines 6-11.

It is necessary that the number of iterations of the repeat-loop of Algorithm 1 is finite, as it would not terminate otherwise. When proving this for Theorem 11, it is needed that equivalence queries yield counterexamples of the form $\sigma\rho$ with $\sigma \in I$. According to the next lemma, this poses no restrictions on the existence of counterexamples.

Lemma 10 Let $A, A' \in \mathcal{A}$ be automata. There exists a string $\tau \in \Sigma^*$ that distinguishes A and A' if and only if there exist strings $\rho \in \Sigma^*$ and $\sigma \in I$ such that $\sigma\rho$ distinguishes A and A'.

Proof:

$\Leftarrow$) Holds trivially.

$\Rightarrow$) Let τ be a string that distinguishes A and A'. According to Definition 3, $obs_A(\tau) \neq obs_{A'}(\tau)$. From Definition 5, it follows that $\tau \notin I$. Let σ be the longest prefix of τ included in I. This prefix always exists since $\epsilon \in I$. Then a string ρ exists such that $\sigma\rho = \tau$. $\square$

Theorem 11 Let $H, H' \in \mathcal{A}$ be two subsequent hypotheses in the execution of Algorithm 1. The repeat-loop iterates a finite number of times for these two hypotheses.

Proof: Let τ be the counterexample of line 3 of Algorithm 1. According to Lemma 10, then also a counterexample $\sigma\rho$ with $\sigma \in I$ exists, which can be constructed as in the proof. Take $\sigma_1\rho_1$ as the I-minimal distinguishing string of line 5.

If $\sigma_1\rho_1$ distinguishes H and L (and not H' and L), the body of the inner loop will never be executed, thus it iterates a finite number of times.

If $\sigma_1\rho_1$ distinguishes H' and L, $\sigma_1\rho_1$ is used as counterexample to construct H'' from H'. Subsequently, an I-minimal distinguishing string $\sigma_2\rho_2$ is found for H and H''. As H' was constructed from H for counterexample $\sigma\rho$, it holds that $obs_L(\sigma\rho) \neq obs_H(\sigma\rho)$. When also taking into account that H'' was constructed from H', it holds that $obs_L(\sigma\rho) = obs_{H'}(\sigma\rho) = obs_{H''}(\sigma\rho)$. Therefore, $\sigma\rho$ distinguishes H and H''. As ρ_2 is minimal, it holds that $|\rho_2| \leq |\rho|$.

Furthermore, $\rho_2 \neq \rho_1$, because H'' was constructed such that $obs_{H''}(\sigma_1\rho_1) = obs_L(\sigma_1\rho_1)$. Also, $|\rho_2| \geq |\rho_1|$, because ρ_1 is minimal. Since only a finite number of strings have a length between $|\rho_1|$ and $|\rho|$, and each iteration of the inner loop will find a new I-minimal distinguishing string, the loop will only iterate a finite number of times. $\square$

Using I-minimal distinguishing strings as counterexample can be a big advantage, as equivalence queries can be time consuming. Additionally, they are not random counterexamples, as they consist of a prefix from I and a minimal length suffix. However, obtaining an I-minimal string that distinguishes H and H' is not trivial. Algorithms for doing this are given in the next section.

3 Algorithms for finding I-minimal distinguishing strings

Figure 1 gives an overview of the algorithms that will be explained in this section. By chaining the algorithms after each other, according to the arrows, an I-minimal distinguishing string can be found for two hypotheses. The labels of the arrows indicate which result from the previous algorithm is passed to the next algorithm.

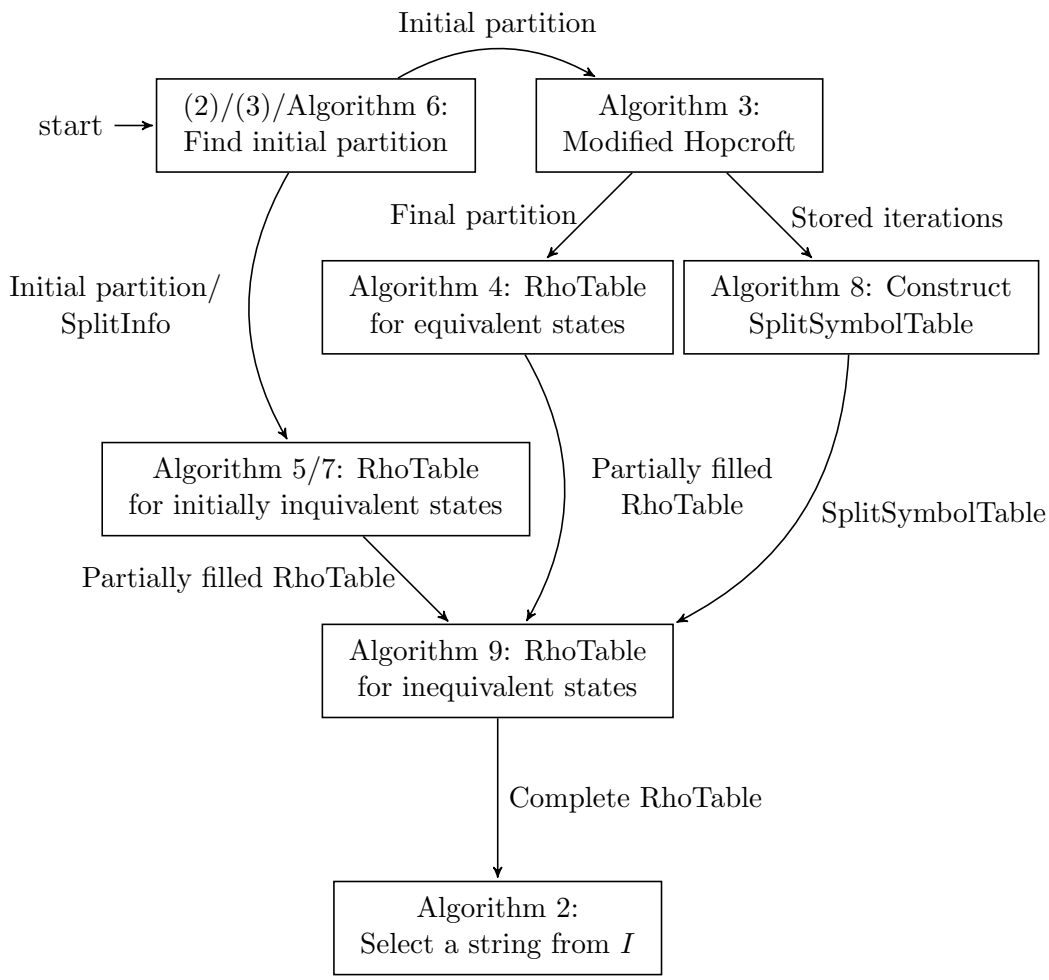

Figure 1: Overview of algorithms for finding I-minimal distinguishing strings

3.1 Selecting a string from I

Let $H = (Q, \Sigma, q_0, D, \delta, obs)$ and $H' = (Q', \Sigma, q_0', D', \delta', obs')$ be two hypotheses. An I-minimal string, distinguishing H and H', can be found by searching for a $\sigma \in I$ that results in the shortest distinguishing string ρ starting in the states $\delta(q_0, \sigma)$ and $\delta'(q_0', \sigma)$. Hence, an I-minimal distinguishing string can be found by Algorithm 2, if it is feasible to calculate a table 'RhoTable' containing a shortest distinguishing string for every pair of

states $(s, s') \in Q \times Q'$ (see Definition 4). It will be shown subsequently that this can be done. For equivalent states, RhoTable will contain a value 'equivalent', having an infinite length.

Algorithm 2 takes $\rho = \text{RhoTable}[q_0][q'_0]$ as an initial distinguishing string, to which other strings ρ' are compared, since it always holds that $\epsilon \in I$. Furthermore, the shortest user log $\sigma \in I$ is selected if the lengths of the corresponding distinguishing strings ρ are the same.

Algorithm 2: Selecting a string $\sigma \in I$ for a shortest string ρ

Input: Automata $H = (\Sigma, Q, q_0, D, \delta, obs)$ and $H' = (\Sigma, Q', q'_0, D, \delta', obs')$
Input: The RhoTable from Algorithm 9 for inputs H and H'

```
σ := ϵ;
ρ := RhoTable[q0][q'0];
for σ' ∈ I\{ϵ} do
    s := δ(q0, σ);
    s' := δ'(q'0, σ);
    ρ' := RhoTable[s][s'];
    if |ρ'| < |ρ| or |ρ'| = |ρ| and |σ'| < |σ| then
        ρ := ρ';
        σ := σ';
    end
end
```

Result: An I-minimal string $\sigma\rho$ distinguishing H and H'

3.2 Partitioning states

A modification of the algorithm of Hopcroft (1971) is used to find a partition of equivalent states. Remember that a partition of some set Q consists of pairwise disjoint and non-empty subsets of Q, such that the union of those subsets is Q itself. The elements of a partition are called blocks. A partition P refines a partition P' if any block of P is contained in a block of P'.

While constructing the partition, information is obtained from which distinguishing strings for pairs of states can be constructed, as inequivalent states are put in different blocks of the partition. This information will be processed by subsequent algorithms.

Originally, Hopcroft's algorithm is used to partition the states of one automaton. However, here it used to do that for two automata. Hence, the hypotheses need to be joined somehow. Furthermore, Hopcroft's algorithm only uses the input alphabet Σ, the states in Q and the transition function δ of an automaton. These are exactly the elements of a Labelled Transition System. Hence, only the union of the underlying Labelled Transition Systems of the two automata needs to be provided to the algorithm. Definition 12 defines this union formally.

Definition 12

- A Labelled Transition System (LTS) is defined as a 3-tuple (Σ, Q, δ), with its elements defined as in Definition 1
- Let $H = (\Sigma, Q, \delta)$ and $H' = (\Sigma, Q', \delta')$ be two Labelled Transition Systems with $Q \cap Q' = \emptyset$. The union Labelled Transition System of H and H' is defined as $H'' = (\Sigma, Q \cup Q', \delta \cup \delta')$, where $\delta \cup \delta'$ is defined as follows:

  Let $s \in Q \cup Q'$ and $\upsilon \in \Sigma$. Then:

$$(\delta \cup \delta')(s, \upsilon) = \begin{cases} \delta(s, \upsilon) & \text{if } s \in Q \\ \delta'(s, \upsilon) & \text{if } s \in Q' \end{cases} \tag{1}$$

Definition 13 defines the notion of a stable partition. It is based on the definition of Paige and Tarjan (1987). Hopcroft's algorithm refines an initial partition P_0, to find a coarsest stable partition.

Definition 13 Let $H = (\Sigma, Q, \delta)$ be a Labelled Transition System. Let $A \subseteq Q$ be a set of states, and $\upsilon \in \Sigma$ an input symbol. Then the set $X = \{s' | \delta(s', \upsilon) \in A\}$ contains the entering states of A for υ. A partition P is called *stable* with respect to A and υ if for all sets $Y \in P$ it holds that either $Y \subseteq X$ or $Y \cap X = \emptyset$.

Hopcroft's algorithm refines a partition by determining the entering states, i.e. the set X of Definition 13, to some block A in the partition, for each symbol $\upsilon \in \Sigma$. Secondly, for all blocks Y in the current partition, it is checked whether it both contains entering and non-entering states. If this is the case, Y is split in a block solely containing the entering states $(X \cap Y)$, and a block solely containing the non-entering states $(Y \backslash X)$. Note that exactly in this case, the partition is not stable. The block A is then called *splitter*, and the symbol υ the *split symbol*. We say that two states *are split*, if one of the states was an entering state, and the other one not.

The set S contains the blocks that are used as splitter. From Corollary 10 of (Knuutila, 2001) it follows that the initial set of splitters can be the initial partition, but with one block left out. Preferably the biggest one, since that one will cost the most computation in the execution of Algorithm 3, as probably the set of entering states will then be bigger.

If a block of the partition is split into two blocks, it is not necessary to use all blocks as splitters. Let X be a set of entering states for some $A \in S$, and let $Y \in P$. If it holds that $Y \in S$, then according to Lemma 8 of (Knuutila, 2001) Y can be replaced by $Y \cap X$ and $Y \backslash X$. If $Y \notin S$, it suffices to add only one of the blocks $Y \cap X$ and $Y \backslash X$ to S because of Corollary 10 of (Knuutila, 2001).

A modified version of Hopcroft's algorithm is shown in Algorithm 3. The core of the algorithm has remained unchanged, only two small changes were made:

- The splitters are processed in the order they were added to S.
- The history of the execution is saved.

Algorithm 3: Modified algorithm of Hopcroft

```
Input: A Labelled Transition System H = (Σ, Q, δ)
Input: An initial partition P
/* The current time                                                    */
t := 0;
/* The initial set of splitters has time of birth 0                    */
S := {(Y, t) | Y ∈ (P\{Biggest block in P})};
/* Storage for iteration values                                        */
I := [];
while S not empty do
    Remove a set A with the smallest time of birth from S;
    for υ ∈ Σ do
        X := ∅;
        /* Fill X with the entering states of A for υ                  */
        for s ∈ A do
            X := X ∪ {s'|δ(s', υ) = s};
        end
        for Y ∈ P do
            if X ∩ Y ≠ ∅ ∧ Y\X ≠ ∅ then
                t++;
                I[t] = (υ, X ∩ Y, Y\X);
                /* Split a set of the partition                        */
                P := P\{Y};
                P := P ∪ {X ∩ Y, Y\X};
                /* Update the set of splitters                         */
                if Y ∈ S then
                    S := S\{(Y, t')};
                    S := S ∪ {(X ∩ Y, t), (Y\X, t)};
                else /* only one of the two sets is a splitter          */
                    if |X ∩ Y| ≤ |Y\X| then
                        S := S ∪ {(X ∩ Y, t)};
                    else
                        S := S ∪ {(Y\X, t)};
                    end
                end
            end
        end
    end
end
Result: The coarsest stable partition that refines the initial partition P
```

Hopcroft's original algorithm does not assume any order of processing the splitters in S. Hence, random order would be fine. But for the correctness proof in Section 4.1 to work, it is needed that splitters are handled in the order in which they were added to S, i.e. in the order of the splitter's time of birth, as defined in Definition 14. Consequently, the set of splitters S can be implemented as a FIFO queue.

Definition 14 Let A be a splitter in an execution of Algorithm 3. The *time of birth* of A, denoted with $\text{tob}(A)$, is defined as the number of iterations in which a split occurred up to the moment A was added to S. From Definition 15 it follows that the number of iterations in which a split occurred is the same as the number of stored iterations. This number is simply obtained by keeping a counter t that starts at 0 (line 1) and increments with 1 each time a split occurs (line 13).

Definition 15

- An *iteration* of Algorithm 3 is the execution of the code in the inner for-loop on lines 12-27, for some set X, which is constructed on lines 7-10 for some symbol $\upsilon \in \Sigma$ and splitter $A \in S$.
- A *stored iteration* is defined as the 3-tuple containing the value of the variable υ, and the two values of the expressions $X \cap Y$ and $Y \backslash X$, of one iteration. The values are collected on line 14.

The name 'stored iteration' is no coincidence; it is called that way because it indeed needs to be stored. The information in a stored iteration is used later to construct distinguishing strings for pairs of states.

Example 16 As an example, Algorithm 3 will be applied on the states of two Deterministic Finite Automata. These DFAs were also used in (Smetsers et al., 2014). They are shown in Figure 2.

Take $\{\{1,3\},\{2,4,5\}\}$ as initial partition and $\{\{1,3\}\}$ as the set of splitters. The entering states of {1,3} for a are {2,4}. For {2,4,5} from P it holds that $\{2,4\} \cap \{2,4,5\} \neq \emptyset \wedge \{2,4,5\} \backslash \{2,4\} \neq \emptyset$, so it is split into {2,4} and {5}. The set {5} is added to S. In the second iteration, the entering states of {1,3} for b are determined, namely {2,4}. The condition on line 13 holds for no set in P, since it exactly contains {2,4}, thus no split is made. The other iterations are done in the same way. In the last iteration, no $Y \in P$ satisfies the condition on line 13, and S is empty, thus the algorithm stops.

In Figure 3 all iterations of the algorithm are shown, including the not stored iterations. Also the values of the variables and expressions, which are not part of a stored iteration, are shown to give a complete view of the execution of Algorithm 3 on the hypotheses of Figure 2.

From the partition resulting from Algorithm 3, we already know which states are equivalent. Hence, we can put 'equivalent' in the cells of RhoTable for these pairs of states. This is done by Algorithm 4.

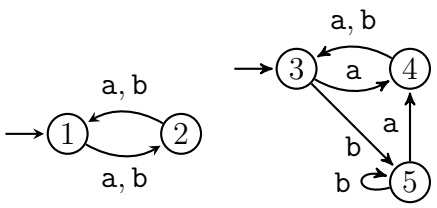

Figure 2: The subsequent hypotheses H (left) and H' (right)

P	S	v	A	X	Y	$X \cap Y$	$Y \backslash X$
$\{\{1,3\},\{2,4,5\}\}$	$\{\{1,3\}\}$	a	$\{1,3\}$	$\{2,4\}$	$\{2,4,5\}$	$\{2,4\}$	$\{5\}$
$\{\{1,3\},\{2,4\},\{5\}\}$	$\{\{5\}\}$	b	$\{1,3\}$	$\{2,4\}$	no split	no split	no split
$\{\{1,3\},\{2,4\},\{5\}\}$	$\{\{5\}\}$	a	$\{5\}$	$\emptyset$	no split	no split	no split
$\{\{1,3\},\{2,4\},\{5\}\}$	$\emptyset$	b	$\{5\}$	$\{3,5\}$	$\{1,3\}$	$\{3\}$	$\{1\}$
$\{\{1\},\{3\},\{2,4\},\{5\}\}$	$\{\{1\}\}$	a	$\{1\}$	$\{2\}$	$\{2,4\}$	$\{2\}$	$\{4\}$
$\{\{1\},\{3\},\{2\},\{4\},\{5\}\}$	$\{\{2\}\}$	b	$\{1\}$	$\{2\}$	no split	no split	no split
$\{\{1\},\{3\},\{2\},\{4\},\{5\}\}$	$\{\{2\}\}$	a	$\{2\}$	$\{1\}$	no split	no split	no split
$\{\{1\},\{3\},\{2\},\{4\},\{5\}\}$	$\emptyset$	b	$\{2\}$	$\{1\}$	no split	no split	no split

Figure 3: The iterations of Algorithm 3 on the hypotheses in Figure 2

Algorithm 4: Filling RhoTable for pairs of equivalent states

```
Input: A partition P
for Y ∈ P do
    for s ∈ Y ∩ Q do
        for s' ∈ Y ∩ Q' do
            RhoTable[s][s'] := equivalent;
        end
    end
end
Result: For each equivalent pair (s, s') ∈ Q × Q' RhoTable[s][s'] contains the value
        'equivalent'
```

3.3 Initial partition

In Example 16, the initial partition and initial set of splitters were not chosen arbitrarily. Algorithm 3 will only produce a partition of equivalent states when the right initial partition and set of splitters are chosen. When states are put in different blocks of the initial partition, they clearly should be inequivalent.

The outputs of DFAs and Moore machines are linked to their states. Thus states that produce different outputs are inequivalent, and can be put in different blocks of the initial partition. Moreover, the string ϵ, which is the shortest possible string, distinguishes those states. Algorithm 3 then does the rest of the work, because it will split states that end up in states with different outputs for the same input sequence.

Mealy machines produce their outputs by taking a transition. Hence, an input sequence that distinguishes two states must be at least one input symbol long. Therefore, an initial partition can be found by putting two states s and s' in the same block when $\lambda(s, \upsilon) = \lambda'(s', \upsilon)$ for all $\upsilon \in \Sigma$. Algorithm 3 will then again do the rest of the work, because it will split states that end up in states that are in different blocks of the initial partition.

For constructing an initial partition, we need to distinguish between automata that produce outputs on their states and automata that produce outputs on their transitions. In all other parts of this thesis we do not need to make this distinction. Therefore, Definition 17 provides a general way to designate states that are in the same block of the initial partition.

Definition 17 Let $(\Sigma, Q, q_0, D, \delta, obs)$ and $(\Sigma, Q', q_0', D, \delta', obs')$ be two automata. Let $s \in Q$ and $s' \in Q'$ be two states. Then s and s' are *initially inequivalent* if there exists a shortest string $\tau \in \Sigma \cup \{\epsilon\}$ that distinguishes s and s'.

With this definition it can be proven that an initial partition of initially equivalent states results in a final partition of equivalent states, when Algorithm 3 is applied on such an initial partition. Lemma 22 establishes that two states that are put in a different block of the final partition, if there exists a distinguishing string, i.e. if the states are inequivalent. The other case is established in the lemma below.

Lemma 18 If Algorithm 3 puts two states s and s' in the same block of the final partition, then they are equivalent.

Proof: The initial partition is constructed, such that equivalent states are put in the same block. Subsequently, Algorithm 3 only refines blocks of that partition, if that block makes the partition not stable. It separates states that are entered from different blocks of the partition for some input symbol. The fact that s and s' are in the same block of the partition, means that for all input symbols υ, $t = \delta(s, \upsilon)$ and $t' = \delta(s', \upsilon)$ were element of the same block. Now assume by induction that t and t' are equivalent. Let $\tau, \tau' \in \Sigma^*$ be accessing strings of s and s' respectively. Then $\tau\upsilon$ and $\tau'\upsilon$ are accessing strings of t and t'. As t and t' are equivalent, $obs(\tau\upsilon\rho) = obs(\tau'\upsilon\rho)$. Since υ was chosen arbitrarily, $\upsilon\rho \in \Sigma^*$, and hence s and s' are equivalent. $\square$

The construction of the initial partition for DFAs and Moore machines is straightforward. For two DFAs $(\Sigma, Q, q_0, F, \delta)$ and $(\Sigma, Q', q_0', F', \delta')$, the initial partition is:

$$\{Q \backslash F \cup Q' \backslash F', F \cup F'\} \backslash \{\emptyset\}. \tag{2}$$

For two Moore machines $(\Sigma, \Gamma, Q, q_0, \delta, \omega)$ and $(\Sigma, \Gamma, Q', q_0', \delta', \omega')$ it is

$$(\bigcup_{\gamma \in \Gamma} \{\{s \in Q | \omega(s) = \gamma\} \cup \{s \in Q' | \omega'(s) = \gamma\}\}) \backslash \{\emptyset\}. \tag{3}$$

For DFAs and Moore machines, the string ϵ can be added to RhoTable for all pairs of states in different blocks of the initial partition. This is shown in Algorithm 5.

Algorithm 5: Filling Rhotable for initially equivalent states of DFAs and Moore machines

Input: The initial partition P_0
Input: Automata $H = (\Sigma, Q, q_0, D, \delta, obs)$ and $H' = (\Sigma, Q', q_0', D, \delta', obs')$ that produce outputs on their states

```
for X ∈ P_0 do
    for Y ∈ P_0, with X ≠ Y do
        for s ∈ X ∩ Q do
            for s' ∈ Y ∩ Q' do
                RhoTable[s][s'] := ϵ;
            end
        end
    end
end
```

Result: A partially filled Rhotable, containing a shortest distinguishing string for all pairs of initially inequivalent states

Algorithm 6 describes how an initial partition can be constructed for Mealy machines. It uses the composed function $\lambda \cup \lambda'$. It can be defined the same way as $\delta \cup \delta'$ in Definition 12.

Algorithm 6 is a partition refinement algorithm, like Algorithm 3. The algorithm starts with the partition of one block containing all states. The partition is then refined by replacing a block Y of the partition by the non-empty blocks in O. The partition O contains a block for each output $\gamma \in \Gamma$. States are in the same block of O when they both output γ for an input symbol $\upsilon \in \Sigma$. The partition O can be implemented as an array that contains an element for each output, such that states with some output can be put in the corresponding array element.

The set 'SplitInfo' keeps track of the splits that occurred. It saves which input symbol resulted in the partition O'. Algorithm 7 uses this to store a shortest distinguishing string in the RhoTable for initially inequivalent states of Mealy machines.

3.4 Splitting tree

The process of refinement in Algorithm 3 can be seen as a tree of which the nodes contain sets of states. The root node contains all the states. It branches to nodes containing the blocks of the initial partition. When some block is split by Algorithm 3, the node containing that block branches to two new nodes containing the two blocks resulting from the split. The leaf nodes contain the blocks of the final partition. The split symbols are put as labels on the edges between the nodes. The tree corresponding to the stored iterations in Figure 3 is shown in Figure 4.

Let s and s' be two initially equivalent but inequivalent states. The *split node* of s and s' is the node that contains the smallest set of states in the tree that contains both s and s'. Hence, the label on the edge between the split node of s and s', and one of the child

Algorithm 6: Constructing an initial partition for Mealy machines

Input: Mealy machines $(\Sigma, \Gamma, Q, q_0, \delta, \lambda)$ and $(\Sigma, \Gamma, Q', q_0', \delta', \lambda')$ with $Q \cap Q' = \emptyset$

```
P := {Q ∪ Q'};
SplitInfo = ∅;
for υ ∈ Σ do
    for Y ∈ P do
        Construct a partition O = ⋃_{γ∈Γ}{⋃_{s∈Y}{s|(λ ∪ λ')(s, υ) = γ}};
        O' = {o ∈ O|o ≠ ∅};
        P := P\{Y} ∪ O';
        SplitInfo = SplitInfo ∪ {(O', υ)};
    end
end
```

Result: The partition P of initially equivalent states, and the set SplitInfo

Algorithm 7: Filling Rhotable for initially equivalent states of Mealy machines

Input: Mealy machines $H = (\Sigma, \Gamma, Q, q_0, \delta, \lambda)$ and $H' = (\Sigma, \Gamma, Q', q_0', \delta', \lambda')$

Input: The set SplitInfo from Algorithm 6 for inputs H and H'

```
for (O, υ) ∈ SplitInfo do
    for X ∈ O do
        for Y ∈ O, with X ≠ Y do
            for s ∈ X ∩ Q do
                for s' ∈ Y ∩ Q' do
                    RhoTable[s][s'] := υ;
                end
            end
        end
    end
end
```

Result: A partially filled Rhotable, containing a shortest distinguishing string for all pairs of initially inequivalent states

nodes must be the symbol for which Algorithm 3 split the two states. The split symbols will be used to construct the distinguishing strings for inequivalent states, as a split means that the partition was not stable for that symbol.

The split symbols of pairs of states can be retrieved from the stored iterations of Algorithm 3. They will be put in a table 'SplitSymbolTable'. Algorithm 8 describes the construction of the SplitSymbolTable in pseudo code. Only split symbols of pairs $(s, s') \in Q \times Q'$ need to be retrieved, as the RhoTable only stores distinguishing strings for those pairs. Each iteration contains two sibling nodes I and D, and the split symbol υ of the edge to their parent node. Each pair $(s, s') \in I \times D$ was split by υ. Hence, this symbol

is stored at SplitSymbolTable[s][s'].

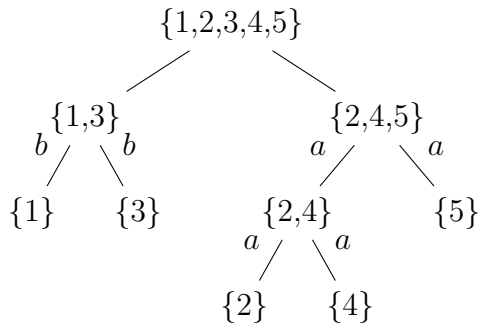

Figure 4: Partition tree of the hypotheses in Figure 2

Algorithm 8: Constructing a table of split symbols for pairs of inequivalent states

Input: Automata $H = (\Sigma, Q, q_0, D, \delta, obs)$ and $H' = (\Sigma, Q', q_0', D, \delta', obs')$
Input: The stored iterations from Algorithm 3 for the inputs: union of underlying LTSes of H and H', and a initial partition for H and H'

```
for (υ, I, D) ∈ stored iterations do
    for s ∈ I do
        for s' ∈ D do
            if s ∈ Q ∧ s' ∈ Q' then
                SplitSymbolTable[s][s'] := υ;
            else if s ∈ Q' ∧ s' ∈ Q then
                SplitSymbolTable[s'][s] := υ;
        end
    end
end
```

Result: For each pair $(s, s') \in Q \times Q'$ of initially equivalent but inequivalent states, SplitSymbolTable contains the split symbol of s and s'

Example 19 Algorithm 8 would construct the table in Figure 5 from the stored iterations from Example 16. This table indeed corresponds with the partition tree in Figure 4.

	3	4	5
1	b	-	-
2	-	a	a

Figure 5: Table obtained by Algorithm 8 using the results of Example 16

3.5 Distinguishing strings for inequivalent states

From the SplitSymbolTable, the distinguishing strings of inequivalent states can be constructed. Let $s \in Q$ and $s' \in Q'$ be two initially equivalent but inequivalent states. Then s and s' have a split symbol υ. The states $t = \delta(s, \upsilon)$ and $t' = \delta'(s', \upsilon)$ must also be inequivalent, as otherwise s and s' would be equivalent. If t and t' are initially inequivalent, RhoTable contains a shortest distinguishing string τ for t and t'. Then $\upsilon\tau$ is a shortest distinguishing string for s and s'. Otherwise the computation can continue by using the transition functions on t, t' and their split symbol. Algorithm 9 describes this in pseudo code.

While the algorithm searches for a shortest distinguishing string for s and s',it also finds one for t and t', and one for all encountered successors. If ρ is the shortest distinguishing string found for s and s', then tail(ρ) is a shortest distinguishing string for t and t'. For completeness, this function is defined in Definition 20. Furthermore, a distinguishing string may already have been found for some pair that is encountered while searching, which means we can simply append it to the prefix that is found up to that moment. Algorithm 9 uses this to avoid doing more work than necessary.

Definition 20 Let τ be a string. If $\tau = \epsilon$, $\text{tail}(\tau) = \epsilon$. Else, let $\tau_0, \tau_1, \ldots, \tau_m$ be the characters of τ. Then the function tail on τ is defined as follows:

$$\text{tail}(\tau) = \begin{cases} \tau_1 \ldots \tau_m & \text{if } m > 0 \\ \epsilon & \text{if } m = 0 \end{cases} \tag{4}$$

Example 21 Consider the iteration of Algorithm 9 in which s is state 1 of Figure 2, and s' state 3. The states 1 and 3 are inequivalent, according to the final partition found in Example 16. Therefore, RhoTable is empty, and SplitSymbolTable contains an input symbol, namely b, according to Example 19. The new states for input b are computed: $s = \delta(1, b) = 2$ and $s' = \delta(3, b) = 5$. The states 2 and 5 are both rejecting, thus initially equivalent, so another round is needed. The split symbol of 2 and 5 is a. Compute again the new states: $s = \delta(2, a) = 1$ and $s' = \delta(5, a) = 4$. Now state 1 is accepting and state 4 rejecting, which means that RhoTable[1][4] $= \epsilon$. Hence, we are done and found RhoTable[1][3] $= ba$ and RhoTable[2][5] $= a$.

Algorithm 9: Finding distinguishing strings for pairs of states

```
Input: Automatons H = (Σ, Q, q_0, D, δ, obs) and H' = (Σ, Q', q'_0, D, δ', obs')
Input: The SplitSymbolTable from Algorithm 8 for inputs H and H'
for s ∈ Q do
    for s' ∈ Q' do
        if RhoTable[s][s'] is empty then
            ρ := ϵ;
            i := 0;
            P[i] := (s, s');
            /* Find ρ and store taken path from (s, s') */
            while RhoTable[s][s'] is empty do
                i++;
                υ := SplitSymbolTable[s][s'];
                s := δ(s, υ);
                s' := δ'(s', υ);
                Append υ to ρ;
                if RhoTable[s][s'] is filled then
                    Append RhoTable[s][s'] to ρ;
                    break;
                else
                    P[i] := (s, s');
                end
            end
            /* Store ρ for (s, s') and its suffixes for the states of the
               path */
            i := 0;
            repeat
                Let (s, s') = P[i];
                i++;
                RhoTable[s][s'] := ρ;
                ρ := tail(ρ);
            until ρ == ϵ;
        end
    end
end
Result: The table RhoTable containing a shortest distinguishing string for each
        pair of inequivalent states (s, s') ∈ Q × Q'
```

4 Algorithm analysis

4.1 Correctness

The algorithms described in the previous section should yield a minimal distinguishing string for each pair of inequivalent states. First, it is proven in Lemma 22 that they yield distinguishing strings. After that, it is proven in Theorem 23 that those strings are minimal.

Lemma 22 Let $(\Sigma, Q, q_0, D, \delta, obs)$ and $(\Sigma, Q', q_0', D, \delta', obs')$ be two automata. Let $(s, s') \in Q \times Q'$ be a pair of inequivalent states. Let ρ be the string that is stored in RhoTable$[s][s']$. Then ρ distinguishes s and s'.

Proof: If s and s' are initially inequivalent, there exists a shortest string $\tau \in \Sigma \cup \{\epsilon\}$ that distinguishes s and s', according to Definition 17. Then $\tau = \rho$ must hold, as τ would have been stored in the RhoTable by Algorithm 5 or Algorithm 7.

If s and s' are initially equivalent, Algorithm 9 constructed ρ, by appending some distinguishing string ρ_s that was already in RhoTable for states t and t', to a string ρ_p consisting of symbols from the SplitSymbolTable. Assume that ρ_s distinguishes t and t'. This is justified, as RhoTable only consists of distinguishing strings for initially inequivalent states, before the execution of Algorithm 9, hence this proof could first have been applied on t, t' and ρ_s.

Let τ, τ' be the accessing strings of s and s'. Then $\tau\rho_p$ and $\tau'\rho_p$ are accessing strings of t and t'. It follows from Definition 4 that $obs(\tau\rho_p\rho_s) \neq obs'(\tau'\rho_p\rho_s)$. Consequently, ρ distinguishes s and s'. $\square$

Theorem 23 Let $(\Sigma, Q, q_0, D, \delta, obs)$ and $(\Sigma, Q', q_0', D, \delta', obs')$ be two automata. Let $(s, s') \in Q \times Q'$ be a pair of inequivalent states. Let ρ be the string that is stored in RhoTable$[s][s']$. Then it must hold that ρ is minimal, i.e. there exists no distinguishing string τ with $|\tau| < |\rho|$.

Proof: When s and s' are initially inequivalent, there exists no shorter string than ρ, according to Definition 17. It was explained in Section 3.3 why that was the case.

Now assume that s and s' are initially equivalent, and that there exists a distinguishing string τ with $|\tau| < |\rho|$. When a contradiction is found, this assumption cannot be true, hence τ cannot exist. In accordance with Definition 24, let $\tau_0, \ldots, \tau_n$ be the symbols of τ, $t_0, \ldots, t_{n+1}$ the path for τ starting in s and $t_0', \ldots, t_{n+1}'$ the path for τ starting in s'.

Either t_n and t_n', or t_{n+1} and t_{n+1}' must be initially inequivalent, as either a symbol $\upsilon \in \Sigma$ distinguishes t_n and t_n', or ϵ distinguishes t_{n+1} and t_{n+1}'. For this proof it will not matter which is the case, so assume that t_n and t_n' are initially inequivalent.

For $0 < i < n$, define A_{i+1} to be the splitter of t_i and t_i'. By induction it follows that those splitters exist. The splitter A_n must be a set from the initial set of splitters, as t_{n-1} and t_{n-1}' are entering states of t_n and t_n', respectively. Furthermore, splitter A_{i+1} will

cause splitter A_i to be added to the set of splitters, if it was not present in that set yet. Consequently, A_1 is the splitter that splits $s = t_0$ and $s' = t'_0$.

Assume for $1 < i < n$, that splitter A_{i+1} used symbol τ_i as split symbol. This is justified, because Lemma 25 tells that choosing another symbol results in a distinguishing string of the same length as $\tau_i \dots \tau_n$. It follows that Algorithm 9 finds $\tau_1 \dots \tau_n$ as distinguishing string for t_1 and t'_1.

Without loss of generality, it can be assumed that $\rho_0 \neq \tau_0$, since otherwise the same proof could be given for tail(ρ) and tail(τ), as for ρ and τ. As ρ_0 is the split symbol of s and s', A_1 must have split s and s' for symbol ρ_0, not for τ_0. Again, it follows from Lemma 25, that τ and ρ have the same length. This is a contradiction with the assumption $|\tau| < |\rho|$, thus no shorter distinguishing string exists. □

Definition 24 Let $(\Sigma, Q, q_0, D, \delta, obs)$ be an automaton. Let $s \in Q$ be a state and $\tau \in \Sigma^+$ a non-empty string. Let $\tau_0, \dots, \tau_k$ be the symbols of τ, i.e. $\tau = \tau_0 \dots \tau_k$ and for $0 \geq i \geq k, \tau_i \in \Sigma$. Then $s_0, \dots, s_{k+1}$ is the path of τ starting in s if and only if $s = s_0$ and for all $0 \geq i \geq k$, $\delta(s_i, \tau_i) = s_{i+1}$.

Lemma 25 Let $(\Sigma, Q, q_0, D, \delta, obs)$ and $(\Sigma, Q', q'_0, D, \delta', obs')$ be two automata. Let $s \in Q$ and $s' \in Q'$ be states which are split by some splitter A at some point in the execution of Algorithm 3. If A can split s and s' for two different symbols a and b, then the corresponding distinguishing strings $a\rho$ and $b\rho'$, found by Algorithm 9, have the same length.

Proof: Assume that A is the first splitter which can make a split for more than one symbol, such that $\rho = \rho'$. This assumption is justified, because it is possible to choose s and s' such that this holds, and then apply this lemma repeatedly on all splitters, which can make a split for more than one symbol, in order of their time of birth.

Without loss of generality it can be assumed that $\delta(s, a) \in A$. Then either $\delta(s, b) \in A$ or $\delta'(s', b) \in A$. First assume $\delta(s, b) \in A$. Define $B = \{\delta(s, a), \delta(s, b)\} \subseteq A$ and $C = \{\delta'(s', a), \delta'(s', b)\}$ with $A \cap C = \emptyset$. Since the states s and s' are split by A and thus have not been split by a splitter with a lower time of birth, it holds for every splitter D with $\text{tob}(D) < \text{tob}(A)$, that $B \cup C \subseteq D$ or $(B \cup C) \cap D = \emptyset$.

Since A is created as a subset of some handled splitter E, with $\text{tob}(E) < \text{tob}(A)$, and because $B \subseteq A$, it must hold that $B \cup C \subseteq E$. From Algorithm 3 it follows that if some set was a splitter, it was was also a set in the partition at the moment this splitter was added to the set of splitters S. The sets of the intermediate partitions are nodes in the splitting tree. Consequently, A and E are nodes in the splitting tree, such that E is the parent node of A. It follows that E is the split node of B and C.

Node E is a split node created by a splitter F with $\text{tob}(F) < \text{tob}(E) < \text{tob}(A)$. From Lemma 26 it follows that the distinguishing string ρ for states having E as split node, consists of symbols corresponding to splitters with a lower time of birth than F and thus A. Therefore no other distinguishing string than ρ is possible according to the assumption made at the start of this proof. Consequently, s and s' have distinguishing strings $a\rho$ and $b\rho$ which have the same length. The same result follows analogously if $\delta'(s', b) \in A$, i.e. for $B = \{\delta(s, a), \delta'(s', b)\}$ and $C = \{\delta'(s', a), \delta(s, b)\}$. □

Lemma 26 Let s and s' be inequivalent states. Let $\rho = \rho_0, \ldots, \rho_n$ be the (non-empty) string constructed by Algorithm 9. Let A be the splitter corresponding to split symbol ρ_i, with $0 \leq i < n$, and let B be the splitter corresponding to split symbol ρ_j, with $i < j \leq n$, as used in Algorithm 3. Then it holds that $\text{tob}(B) < \text{tob}(A)$.

Proof: By induction on n.

Base case If $n = 0$, this holds trivially, as only one split for one splitter occurred.

Step Assume for strings ρ', with $|\rho'| < |\rho|$, that the splitter A' corresponding to split symbol ρ'_i, has a higher time of birth than the splitter B' corresponding to split symbol ρ'_j, with $i < j$. Take $\rho' = \rho_1, \ldots, \rho_n$. Then it remains to prove that for the splitter A corresponding to ρ_0 and the splitter B corresponding to ρ_1, it holds that $\text{tob}(B) < \text{tob}(A)$.

Let $t = \delta(s, \rho_0)$ and $t' = \delta(s', \rho_0)$. Then either $t \in A$ and $t' \notin A$, or $t \notin A$ and $t' \in A$. Consequently, t and t' already have been split before A was handled. Therefore, B was already handled before A. Since Algorithm 3 handles splitters with a lower time of birth earlier than splitters with a higher time of birth, $\text{tob}(B) < \text{tob}(A)$. □

4.2 Time complexity

Let $(\Sigma, Q, q_0, D, \delta, obs)$ and $(\Sigma, Q', q'_0, D, \delta', obs')$ be two automata. Let $n = \#Q + \#Q'$, and $k = \#\Sigma$.

Lemma 27 Algorithm 2 runs in $\mathcal{O}(\#I)$.

Proof: There is only one for-loop, which loops over the elements of $\#I \backslash \{\epsilon\}$. It is reasonable to assume that applying the δ-function and determining the length of a distinguishing string can be done in $\mathcal{O}(1)$. Hence, Algorithm 2 runs in $\mathcal{O}(\#I)$. □

Lemma 28 Algorithm 3 runs in $\mathcal{O}(kn \log n)$.

Proof: This follows from the running time of the original algorithm, which is $\mathcal{O}(kn \log n)$ (Hopcroft, 1971; Gries, 1973). The adaptations on the algorithm have no influence. The set of splitters S can be stored as a FIFO queue, i.e. elements can be stored at the back of the queue and retrieved form the front of the queue in $\mathcal{O}(1)$. Storing an iteration also can be done in $\mathcal{O}(1)$, since this is a simple assignment. □

Lemma 29 Algorithm 4 and Algorithm 5 both run in $\mathcal{O}(n)$.

Proof: A pair (s, s') is never encountered twice, as a state is only contained in one block of the partition. Therefore, at most n^2 assignments to RhoTable are done. □

Lemma 30 Algorithm 6 runs in $\mathcal{O}(kln)$.

- If H and H' are DFAs, the initial partition (2) can be found in $\mathcal{O}(n)$, assuming that the sets of all states and final states are stored in a suitable data structure. For example, the states in those sets are stored in the same order, such that it is possible to strike out the final states in the set of all states.

- If H and H' are Moore machines, the initial partition (3) can also be found in $\mathcal{O}(n)$, as the output function can be applied on each state to determine it's output, such that it can be put in the block for that output.

- If H and H' are Mealy machines, i.e. $H = (\Sigma, \Gamma, Q, q_0, \delta, \lambda)$ and $H' = (\Sigma, \Gamma, Q', q_0', \delta', \lambda')$, the initial partition can be found by Algorithm 6 in $\mathcal{O}(kln)$.

  Proof: Let $l = \#\Gamma$. Algorithm 6 does the following operations:

  1. The partition O can be determined by looping over all states in Y and determining the output for v. This will cost $\#Y$ steps. Constructing all partitions O for the blocks in P will then cost n steps, as each state is only in one block of a partition.
  2. The partition O' can be determined in l steps, as O contains l blocks. Constructing all partitions O' for all the blocks in P will then cost $l \cdot \#P$ steps.
  3. Removing Y and adding O' can be done in one step, if a suitable data structure is used, such as a linked list. Doing this for all blocks of a partition P will cost $\#P$ steps.
  4. Adding an element to a set can also be done in one step. Doing this for all blocks of a partition P will cost $\#P$ steps.

  For k times, the above operations are done for some partition P. Worst case, P will contain n blocks, such that the second operation costs ln steps. The other operations only cost n steps. Thus Algorithm 6 runs in $\mathcal{O}(kln)$.$\square$

Lemma 31 Algorithm 7 runs in $\mathcal{O}(n^2)$.

Proof: Algorithm 6 splits each pair of states at most once, as a split causes the states to be put in another block of the partition. Thus Algorithm 7 can do at most n^2 assignments to RhoTable. Lemma 32 gives a more explicit proof for a similar algorithm. $\square$

Lemma 32 Algorithm 8 runs in $\mathcal{O}(n^2)$. Let $T(n)$ be the number of pairs of states to which an assignment of an input symbol is done, for n states that were part of some split from the stored iterations. According to Lemma 33, the split of a set that results in the most pairs is one which splits a set evenly, thus this is assumed, in the definition $T(n)$, for $n > 2$:

$$T(n) = \begin{cases} \frac{1}{4}n^2 + 2 \cdot T(\frac{n}{2}) & \text{if } n \text{ is even} \\ \frac{n+1}{2} \cdot \frac{n-1}{2} + T(\frac{n+1}{2}) + T(\frac{n-1}{2}) & \text{if } n \text{ is odd} \end{cases}$$

To prove: $T(n) \leq n^2$ Proof:

Base cases No pair can be constructed from one element, so $T(1) = 0$. Exactly one pair can be constructed from two elements, so $T(2) = 1$. Now it holds that $T(1) \leq 1^2 = 1$ and $T(2) \leq 2^2 = 4$.

Step Induction hypothesis: Assume $T(k) \leq k^2$ for $k < n$. To prove: $T(n) \leq n^2$. Proof:

- If n is even, $T(n) = \frac{1}{4}n^2 + 2T(\frac{n}{2}) \leq \frac{1}{4}n^2 + 2 \cdot (\frac{n}{2})^2 = \frac{1}{4}n^2 + 2 \cdot \frac{1}{4}n^2 = \frac{3}{4}n^2 \leq n^2$
- If n is odd, $T(n) = \frac{n+1}{2} \cdot \frac{n-1}{2} + T(\frac{n+1}{2}) + T(\frac{n-1}{2}) \leq \frac{1}{4}n^2 - \frac{1}{4} + (\frac{n+1}{2})^2 + (\frac{n-1}{2})^2 = \frac{1}{4}n^2 - \frac{1}{4} + \frac{n^2+2n+1}{4} + \frac{n^2-2n+1}{4} = \frac{1}{4}n^2 - \frac{1}{4} + \frac{2n^2+2}{4} = \frac{3}{4}n^2 + \frac{1}{4} \leq n^2$ for $n \geq 1$. $\square$

Lemma 33 A split of a set results in the most pairs, if it splits the set evenly, i.e. a set of n elements is split in two sets of $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ elements, which results in $\lceil \frac{n}{2} \rceil \cdot \lfloor \frac{n}{2} \rfloor$ pairs.

Proof: Suppose a set of n elements is split in a set of m and a set of l elements, with $m+1 < l$, i.e. the set was not split evenly. Then the sets of m and l elements should result a smaller number of pairs than sets of $m+1$ and $l-1$ elements (note that the total number of elements stays the same): $(m+1)(l-1) = ml + l - m - 1 > ml + m + 1 - m - 1 = ml$. $\square$

Lemma 34 Algorithm 9 also is of $\mathcal{O}(n^2)$, or more precise: $\mathcal{O}(ml)$ for $m = \#Q$ and $l = \#Q'$.

Proof: Each pair of states (s, s') is handled once by the two outer for-loops on line 1 and 2. In the body of the those for-loops, a string ρ that distinguishes the two states is found, when RhoTable$[s][s']$ is empty, i.e. when s and s are initially equivalent but inequivalent. Therefore, the body of the inner while loop is executed at least once. In search for a string ρ at most one pair (t, t'), with RhoTable$[t][t']$ filled is encountered, since this satisfies the condition of the if statement on line 13, after which the loop is exited by the break on line 15. Hence, pairs of states can be encountered more than one time, but at most $2ml$ pairs are encountered in total. The operations done for each pair are done in $\mathcal{O}(1)$.

Only the while-loop was considered, since the repeat-until-loop on line 21-26 loops over exactly the same pairs as the while-loop. Assuming that the implementation of the tail-function is of $\mathcal{O}(1)$, this loop does not increase the complexity of the algorithm. Thus Algorithm 9 runs in $\mathcal{O}(ml)$. $\square$

It can be concluded that all algorithms together run in $\mathcal{O}(n^2)$, if

- $\#I \leq n^2$,
- $k \log(n) \leq n$, and
- $kl \leq n$, if Algorithm 6 is used.

In practice, this will most of the time be the case. The number of input sequences in the set I of log files can even be made smaller if it really influences the complexity. Hence, a running time of $\mathcal{O}(n^2)$ should be feasible.

5 Experiments

5.1 Comparing hypotheses

In (Smeenk, 2012), a model was learned of a software component of Océ, called 'Engine Status Manager'. All intermediate hypotheses of this experiment were available. Hence, this offered an opportunity to apply the metric on pairs of subsequent hypotheses, and see if the last hypothesis of a pair could be distinguished by an I-minimal string.

When using $I = \{\epsilon\}$, the last hypothesis of a pair was distinguished for 4 pairs. This is the same number as found in (Smetsers et al., 2014).

Unfortunately, no user logs were directly available. Since it would cost a lot of time to collect suitable logs, it was decided to generate some logs from the information available. As the last hypothesis was known, it was possible to extract some input sequences from this model. The hypotheses were (deterministic) Mealy machines. Consequently, taking some random path in the last hypothesis could result in any input sequence in Σ^*. Therefore, only inputs with a different destination and source state were taken.

Three runs were done for different sets of I. The first run had 30000 logs with lengths between 1 and 50 inputs, the second run 10000 between 1 and 100 inputs, and third run 10000 between 20 and 50 inputs. In total, an I-minimal distinguishing string was found for the last hypothesis of 88 pairs. As there were 135 pairs in total, this happened in 65% of the cases. For the 88 found pairs only 12 strings from the logs were used. For different pairs of hypotheses that were distinguished for the same $\sigma \in I$, often the corresponding ρ was the same, except for the last few symbols. This indicates that if the I-minimal distinguishing string was used to improve the last hypothesis, there would not be so many pairs of which the last hypothesis could be distinguished. Nevertheless, this result indicated that the user log metric can have a significant impact on the learning process.

5.2 Running the adapted learning algorithm

By Smeenk et al. (2015), an advanced testing method, using adaptive distinguishing sequences, was proposed. While the complete model wasn't reached in (Smeenk, 2012), it had been done with this method. Therefore, the testing method was chosen to find counterexamples within the (adapted) learning algorithm. The real world system was not available, but the complete model (made manually) was. Therefore, this model was used as a 'system under learning' in the learning algorithm.

In this setting, seven runs were done. Each time, 100000 logs were generated from the manual model. The generation was done as explained in the previous section. Each run used a union of ten equally big sets of logs having lengths between i and $i + 9$ for $i \in \{1, 11, \ldots, 91\}$. The results of each run are shown in Figure 6.

All adaptations due to inconsistency with a log $\sigma \in I$ were done in the beginning. After that the testing method found most of the counterexamples. On average, the last hypothesis of 2.57 hypothesis pairs were distinguished by an I-minimal string. This is much lower than in the number of the experiment of the previous section. As discussed

there, this can be explained by the fact that similar I-minimal distinguishing strings could be found for several pairs, while in this experiment the next hypothesis is adapted to it.

Due to the randomization used in the testing method, each time other counterexamples and thus other hypotheses are found. This explains the variation in the number of input symbols that needed to be send to the system under learning to perform the learning. However, on average, the adapted learning algorithm needed to send less input symbols than the standard learning algorithm (both using the testing method of (Smeenk et al., 2015)). The average number of input symbols obtained from four runs of the standard algorithm is 489104885. For the adapted one it is 371459071. This is a reduction of 24%.

Number of input symbols	Distinguished by $\sigma \in I$	Distinguished by I-minimal string
276584179	15	4
368574392	21	4
275139665	15	3
285998888	17	3
315423096	16	2
421583792	17	2
656909490	4	0

Figure 6: Runs of adapted learning algorithm. Each row contains the results of one run, namely: the number of input symbols supplied to the system under learning, the number of hypothesis pairs distinguished from the final model by a $\sigma \in I$, and the number of hypothesis pairs of which the last hypothesis was distinguished by an I-minimal string.

6 Related work

A metric that uses lengths of distinguishing strings, applied in an ultrametric space, and used to adapt an active learning algorithm, is based on (Smetsers et al., 2014). The metric of that paper is equivalent with the user log metric in this thesis, if I would only contain ϵ. Furthermore, the paper only considers DFAs and not the more general notion of an automaton as is used here.

The paper did not specify how to find minimal distinguishing strings, but this was done in (Smetsers and Moerman, 2015), which was written concurrently with this thesis. In the paper also the algorithm of Hopcroft (1971) was used to find a partition of equivalent states. As only Mealy machines are considered, the initial partition is constructed while running Hopcroft. In this thesis the construction of the initial partition and running the algorithm of Hopcroft was separated to allow several special cases of automata instead of only Mealy machines. However, their algorithm for constructing minimal distinguishing strings by means of a splitting tree was optimized to reach a time complexity of $kn \log n$.

Another way of finding minimal distinguishing strings is described in (Gill, 1962). This algorithm has time complexity kn^2, and does not use Hopcroft, thus in every step of the calculation that time complexity is really reached. The minimal distinguishing strings are calculated by constructing tables based on accessing sequences and suffixes that make the strings distinguishing. The way in which distinguishing strings are constructed from split symbols is very similar to the way it is done in this thesis. Again, the algorithms only apply to Mealy machines.

As explained in Section 5.2, the adapted learning algorithm can reduce the number of input symbols that need to be processed by the system under learning. Other approaches are certainly possible. For example, by using that the input sequences can be prefixes of input sequences asked later by the learning algorithm (Bauer et al., 2012). Furthermore, the complexity of learning can be reduced by taking system specific features into account (Hungar et al., 2003). Additionally, finding counterexamples for an intermediate model can be hard. In this case, advanced testing methods, such as using adaptive distinguishing sequences (Smeenk et al., 2015), can help a lot.

7 Conclusion

A metric was proposed for measuring distances between intermediate models of an active learning algorithm. The intermediate models can be instances of an automaton, which is basically a Labelled Transition System that produces some kind of output for every possible input sequence. The metric enables to measure if an intermediate model is at least as close to the final model as its predecessor. This was used to adapt the learning algorithm. If an intermediate model is further away than its predecessor, the I-minimal distinguishing string found by the metric is used to improve the model. Otherwise, the learning algorithm continued in the standard way, by searching for a counterexample to construct a new intermediate model.

Algorithms were given to find an I-minimal distinguishing string for two automata. The algorithm of Hopcroft enabled to find a partition of equivalent states, for the states of the two automata. It is an algorithm that refines an initial partition, until the partition is stable, i.e. there is no input sequence such that transitions for that input sequence can be taken from two states of the same block to two states of different blocks.

The initial partition is a courser partition than the partition of equivalent states. Some states that turn out to be inequivalent, are in the same block of the initial partition. The construction of the initial partition depends on the specific kind of automaton that is used to represent the intermediate models. If an automaton produces outputs on its states, the initial partition consists of blocks containing states that produce the same output. If an automaton produces outputs on its transitions, a refinement algorithm is applied to find blocks of states that produce the same output for every input symbol on their outgoing transitions.

By storing the iterations done to construct the partition of equivalent states, it is known which input symbol caused states to be put in a different block of the partition. Hence, these split symbols can be used to construct distinguishing strings for each pair of states. The strings are constructed by taking the transitions of the split symbol from each pair of states, and continue to do this for the reached pair of states, until two states are reached that are each in a different block of the initial partition. From the initial partition, a symbol that distinguishes those states can be retrieved directly.

By taking the transitions for each input sequence in the user logs I, starting in the two initial states of two automata, two 'sensible' states are reached. A string $\sigma\rho$ is chosen as the I-minimal distinguishing string of the two automata, if the user log $\sigma \in I$ leads to a pair of two states for which a shortest distinguishing string ρ was found. The distance between the two automata then is $2^{-|\rho|}$ according to the metric.

A proof of correctness was given, that the found strings ρ are shortest distinguishing strings. Furthermore, it was proven that the total complexity of executing all algorithms is $\mathbb{O}(n^2)$. Here n is the total number of states of the models the metric is applied on.

The applicability, of the theory discussed in this thesis, was shown in the experiments. Several pairs of hypotheses were found, for which the last hypothesis has an I-minimal distinguishing string. Moreover, the experiments indicate that the usage of the adapted learning algorithm results in a substantial reduction of the number of input symbols that

need to be send to the system under learning. In the first experiments on the case study of Océ's the Engine Status Manager, using the testing method of Smeenk et al. (2015) to find counterexamples, a reduction of 24% was reached.

8 Future work

There are several possibilities to continue with the work presented in this thesis. First of all, more experiments can be done to verify if the substantial reduction of inputs symbols is indeed around 24%, for the case study of the Engine Status Manager. Besides that, the experiments could be elaborated, by calculating the distance between each intermediate model and the final model. This should give an insight on the effect the metric has on improving the intermediate models, and also the effect of the testing method that is used to find counterexamples. Additionally, different testing methods for finding counterexamples could be tried, to see whether this has any effect on the performance of the user log metric in the adapted learning algorithm. Moreover, the adapted learning algorithm can be applied on other case studies to see if similar results can be obtained as with the case study of the Engine Status Manager.

In the experiments, no real user logs were used, so evidently, the experiments could be repeated with real logs. Furthermore, all user logs are now treated equally, no matter how often they were recorded from users. Hence, the metric could be adapted, such that not only the length of the distinguishing string ρ, is considered, but also the frequency of a log σ in I. Additionally, logs starting with frequent occurring input symbols could be given some kind of priority, because a high frequency suggests that the input symbol has an important effect. Finally, it may be needed to include some techniques of passive learning, if the set of user logs is very big.

References

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing system states by active learning algorithms. In *Eternal Systems*, pages 61–78. Springer, 2012.

Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert E Schapire, and Linda Sellie. Efficient learning of typical finite automata from random walks. *Information and Computation*, 138(1):23–48, 1997.

Arthur Gill. Introduction to the theory of finite-state machines, 1962.

David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2(2):97–109, 1973.

John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, DTIC Document, 1971.

Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *Computer Aided Verification*, pages 315–327. Springer, 2003.

Timo Knuutila. Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250(1):333–363, 2001.

Kevin P Murphy. Passively learning finite automata. Santa Fe Institute, 1995.

Robert Paige and Robert E Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

Wouter Smeenk. Applying automata learning to complex industrial software, 2012.

Wouter Smeenk, Joshua Moerman, D Jansen, and F Vaandrager. Applying automata learning to embedded control software, 2015. *To appear in Proceedings of the 17th International Conference on Formal Engineering Methods, Paris, November 2015.*

Rick Smetsers and Joshua Moerman. Minimal Separating Sequences for All Pairs of States, 2015.

Rick Smetsers, Michele Volpato, Frits Vaandrager, and Sicco Verwer. Bigger is Not Always Better: on the Quality of Hypotheses in Active Automata Learning. *The 12th International Conference on Grammatical Inference*, pages 167–181, 2014.

Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, pages 256–296. Springer, 2011.