Relevant Literature

Wesley van der Lee

August 25, 2017

1 Relevant Literature

1.1 Terminology

This chapter discusses notorious algorithms for inferring state machines. In order to keep consistency between all discussed related work, this section establishes a formal mathematical notation which will be used in the continuance of this document. The notation will be consistent with the notation proposed by Sipser [1].

The model used to formalize state machines is a deterministic finite automaton (dfa) U. An example dfa $\mathcal{A}$ is depicted in Figure 1. As can be seen, a dfa contains states, letters, transitions, a start state and accepting states. A dfa can thus be formalized by a 5-tuple $U = (Q, \Sigma, \epsilon, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \to Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

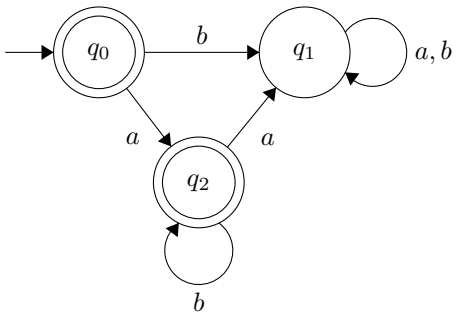

Figure 1: Example dfa $\mathcal{A}$.

The example dfa $\mathcal{A}$ shows three states: q_0, q_1 and q_2 depicted by the circles. Double edged circles indicate that the state is an accepting state, which in the example of $\mathcal{A}$ is the case for q_0 and q_2. Transitions from one state to another are indicated by arrows and their corresponding input letter. From this follows that the input alphabet of $\mathcal{A}$ solely consists of the elements a and b. In general,

a state $q' \in Q$ which is the result of a transition from another state $q \in Q$ with input letter $a \in \Sigma$, is also called the a-successor of q, i.e. $q' = \delta(q, a)$ is the a-successor of q. In the example of dfa $\mathcal{A}$, state q_1 is the b-successor of state q_0. The successor-notation can also be extended for words by defining $\delta(q, \varepsilon) = q$ and $\delta(q, wa) = \delta(\delta(q, w), a)$ for $q \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$. For words $w \in \Sigma^*$ the transition function $\delta(q, w)$ implies the extended transition function: $\delta(q, w) = \delta(q_0, w)$ unless specifically specified. Moreover, by also utilizing the notation of Vazirani et al. [2], a state of a general dfa U can be denoted as $U[w]$ for $w \in \Sigma^*$, where $U[w]$ corresponds to the state in U reached by w, i.e. $U[w] = \delta(q_0, w)$. The singleton transition without an associating input letter, indicates the empty transition, which points to the initial starting state of $\mathcal{A}$.

The example dfa $\mathcal{A}$ from Figure 1, can thus formally be written as $\mathcal{A} = \{Q, \Sigma, \delta, q_0, F\}$ where

1. $Q = \{q_0, q_1, q_2\}$,
2. $\Sigma = \{a, b\}$,
3. δ is described as:

	a	b
q_0	q_2	q_1
q_1	q_1	q_1
q_2	q_1	q_2

4. q_0 is the starting state, and
5. $F = \{q_0, q_2\}$.

Now the dfa can be modeled and formalized, one can establish a set of words over symbols in Σ that also includes the empty word ε which is named Σ^*. The *-notation follows from the unary operation, which works by attaching any number of strings in Σ together: $\Sigma^* = \{x_1, x_2 \ldots, x_{k-1}, x_k | k \leq 0$ and each $x_i \in \Sigma\}$. In $\mathcal{A}$ one can see that word $w_1 = abbb \in \Sigma^*$, because w_1 leads to an accepting state[1]. A word $w_2 = abab$ can be identified as not a member of Σ^*: $w_2 \notin \Sigma^*$[2]. For words $w, w' \in \Sigma^*$, $w \cdot w'$ is a concatenation-operation of the two words. The concatenation-operation of two words can also be written by omitting the operator $\cdot$ and just write ww'.

For a word $w \in \Sigma^*$ the λ-evaluation $\lambda(w)$ returns 1 iff the dfa in question accepts word w, that is if the extended transition function for q_0 concludes in an accepting state. The function $\lambda(w)$ returns 0 if the extended transition function results in a rejecting state.

1.2 Learning Regular Sets from Queries and Counterexamples

Learning Regular Sets from Queries and Counterexamples by Dana Angluin [3] forms the basis of many modern state machine inference algorithms. Her research introduces the polynomial complex L* algorithm for learning a regular set, a task which before was computationally intractable because it was proven to be NP-hard [4]. The basic idea of L* is a learner whose goal is to create a

[1] $\delta(q_0, abbb) = \delta(\delta(q_0, a), bbb) = \delta(\delta(\delta(q_0, a), b), bb) = \delta(\delta(\delta(\delta(q_0, a), b), b), b) = \delta(\delta(\delta(q_1, b), b), b) = \delta(\delta(q_1, b), b) = \delta(q_1, b) = q_2 \in F \rightarrow abbb \in \Sigma*$.

[2] $\delta(q_0, abab) = \delta(\delta(q_0, a), bab) = \delta(\delta(\delta(q_0, a), b), ab) = \delta(\delta(\delta(\delta(q_0, a), b), a), b) = \delta(\delta(\delta(q_2, b), a), b) = \delta(\delta(q_2, a), b) = \delta(q_1, b) = q_1 \notin F \rightarrow abab \notin \Sigma*$.

equitable conjecture by utilization of an expert system, the Minimally Adequate Teacher (MAT). The establishment of a conjecture is achieved by posing two types of queries to the MAT: membership queries and equivalence queries. A membership query inquires a given word w to the MAT, which answers *yes* or *no* depending on whether w is a member of the to be hypothesized System Under Test (SUT): the unknown set U that describes the behavior. This is equivalent to the *lambda*-evaluation for a word w that depicts whether w is recognized by the set U: $\lambda(w) \rightarrow \{0, 1\}$. The learner can also verify a conjecture to the MAT by posing an equivalence query. The MAT then answers *yes* if the conjecture is equal to the set U. If this is not the case, the MAT provides a counterexample, which is a string w' in the symmetric difference of the conjecture and the unknown language.

The learner keeps track of the queried strings, classified by either a member or non-member of the unknown regular set U. This information is organized in an observation table that consists of three fields: a nonempty finite prefix-closed set S of strings, a nonempty finite suffix-closed set E of strings and a finite function T that maps all entries that are formed by concatenating the prefix and suffix together. A word u recognized by the set U can be typically of the form $u = sae$, for $s \in S$, $e \in E$ and $a \in \Sigma$, meaning a word starts with a prefix and ends with a suffix. Sometimes a one-letter extension in Σ is added to the prefix, they identify transitions. The latter will become apparent in the running example of the L* algorithm. Angluin applies the notation of words in the form $u = sae$ as $u \in ((S \cup S \cdot \Sigma) \cdot E)$, thus if $u \in U$ then $T(u)$ will result to 1 and 0 otherwise. Function T thus corresponds to the earlier mentioned λ-function: $T(w) \mathrel{\widehat{=}} \lambda(w)$. To maintain consistency with Angluin's work, the function T notation will be used for the continuance of this section. In conclusion an observation table can be denoted as a 3-tuple (S, E, T).

To clarify the terminology, imagine the example where a learner is required to learn the behavior of dfa $\mathcal{A}$ described in Figure 1. Initially any information except $\mathcal{A}$'s alphabet Σ is unknown to the learner, but the learner has access to a MAT that can answer membership and equivalence queries. The learner starts by first creating the observation table. Set S initially contains the input alphabet of $\mathcal{A}$ and the empty string ε as one-letter prefixes. Thus $S = \{\varepsilon\} \cup \Sigma = \{\varepsilon, a, b\}$ The commencing distinguishing suffix is ε, therefore set $E = \{\varepsilon\}$. The learner then fills the observation table $S \times E$ with the output of membership queries, depending on whether an entry $e \in S \times E$ is accepted by $\mathcal{A}$ or not. This leads to the observation table depicted in Table 1a. Note that the table vertically distinguishes two sections that are separated by an additional line. That is, set S is divided into two subsets. The top section of S represents all distinct rows with respect to the outcome of T. Since $row(\varepsilon) = 1$ and $row(b) = 0$, those distinct rows are put in the top part of S. $Row(a)$ results to 1, which is equivalent to $row(\varepsilon)$, hence $row(a)$ is put in the bottom part of S.

The learner queries for the right amount of data from the MAT, by ensuring that the observation table is both closed and consistent. An observation table is closed if for every entry $e \in S \times \Sigma$, there exists an element $s \in S$ such that $row(e) = row(s)$. If the table is not closed, there exists an entry $e \in S \times \Sigma$ where no prefix will lead to. Table 1a is not closed, because this property is not ensured for the state $\mathcal{A}[b]$. The table clearly depicts at least two states, because there are two distinct rows. However $\mathcal{A}[b]$ requires information on where to transit from that state. Hence the access sequence of the state $\mathcal{A}[b]$ prepended

	ε
ε	1
b	0
a	1

(a)

	ε
ε	1
b	0
a	1
ba	0
bb	0

(b)

	ε	b
ε	1	0
b	0	0
a	1	1
ba	0	0
bb	0	0
aa	0	0
ab	1	1

(c)

Table 1: Gradually growing observation tables corresponding to various steps of the L* algorithm: (a) initial observation table T_0, (b) observation table T_1 that is a closed and consistent version of the initial observation table, (c) final observation table T_2 describing dfa $\mathcal{A}$.

with one-letter extensions are added to the set of prefixes. In other words $\{ba, bb\}$ are to be added to the set S. This results in a bigger observation table, depicted in Table 1b. The learner identifies which suffix distinguishes the rows and adds the word to the set S. An observation table is consistent when for $s_1, s_2 \in S$ such that $row(s_1) = row(s_2)$, if for all a in the alphabet Σ holds that $row(s_1 \cdot a) = row(s_2 \cdot a)$. If the table is not consistent, the language would not be regular as it shows non-deterministic behavior. The learner identifies for which $s_1, s_2 \in S$ distinguishes the result of output T and adds the word to the set E. Given the observation table in 1b, the observation table is consistent.

If (S, E, T) is closed and consistent, an acceptor $H(S, E, T)$ can be defined which is consistent with function T. H then forms the hypothesized model based on the contents of the observation table. A conjecture $H = (\Sigma, Q, \delta, q_0, F)$ can be modeled as follows:

1. $Q = \{row(s) | s \in S_H\}$
2. $q_0 = row(\varepsilon)$
3. $F = \{row(s) | s \in S_H \text{ and } T_H(s, \varepsilon) = 1\}$
4. $\delta(row(s), a) = row(sa)$

A hypothesized dfa H_0 generated according to the steps above consistent to the observation table 1b has the following contents: $Q = \{\mathcal{A}[\varepsilon], \mathcal{A}[b]\}$, $q_0 = \mathcal{A}[\varepsilon]$, $F = \mathcal{A}[\varepsilon]$. Conjecture H_0 is visualized in Figure 2. The learner then verifies the conjecture to the MAT and receives a counterexample back, indicating that the conjecture is inequivalent to U. Although the learner was unable to see this, the model depicted in Figure 2 clearly differs from the SUT shown in Figure 1. Suppose that the learner received the counterexample word $w = ab$, which is a valid counterexample because $\lambda_A(ab) = 1$, $\lambda_{H_0}(ab) = 0$, thus $\lambda_A(ab) \neq \lambda_{H_0}(ab)$ holds.

Analysis of the counterexample $w = ab$ shows that if suffix b is added, H_0 predicts the wrong output. Element b is therefore identified as a distinguishing suffix and hence added to set E. The observation table must now be updated again: the new entries caused by the appearance of the b-column are filled and in order to make the table closed again, new prefixes for the new state $\mathcal{A}[a]$ have to be determined. This results in the observation table depicted in Table

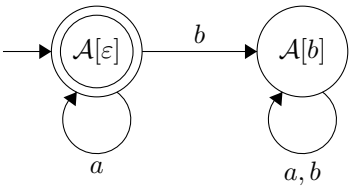

Figure 2: Hypothesized conjecture dfa H_0 corresponding to observation table T_1 (Table 1b).

1c. Following the steps to establish a dfa from an observation table yields the original dfa shown in Figure 1 which is the correct dfa that corresponds to the observation table T_2 shown in Table 1c. After the learner poses an equivalence query with this dfa, the MAT answers *yes*, indicating that the correct dfa has been inferred and the learner can stop learning.

Up until now, the only focus was for the conjecture to be consistent with T. Since a dfa with the state size equal to the number of observations, that is each observation leads to a new case state in a conjecture, such a dfa would be incorrect because it lilely incorrectly models unforeseen future observations. An hypothesized conjecture H should thus not only be consistent with T, but also have the smallest set of states opposed to a non-isomorphic dfa. The L* algorithm only learns a dfa consistent with T and that has the smallest set of states. Angluin proves this as follows. Let q_0 be the starting state ($row(\varepsilon)$) and δ be the transition function from one state to another in the acceptor, then for $s \in S$ and $a \in \Sigma$ holds that because $\delta(row(s), a) = row(s \cdot a)$ follows $\forall s \in (S \cup S \cdot \Sigma) : \delta(q_0, s) = row(s)$, thus the closed property ensures that any row in the observation table corresponds with a valid path in the acceptor. $\forall s \in (S \cup S \cdot \Sigma) \forall e \in E \rightarrow \delta(q_0, s \cdot e)$ is an accepting state if and only if $T(s \cdot e) = 1$, thus due to the consistency with finite function T, a word will be accepted by the acceptor if it is in the regular set. To see that $H(S, E, T)$ is the acceptor with the least states, one must note that any other acceptor H' consistent with T is either isomorphic or equivalent to $H(S, E, T)$ or contains more states.

The algorithm L* is listed in Listing 1.

```
1  S = E = {λ}
2  (S,E,T) ← MQ(λ) ∪ ∀a ∈ A : MQ(a) # (S,E,T) is the observation table
3
4  While M is incorrect: # M is the conjecture
5    While (S,E,T) is not consistent or not closed
6      if (S,E,T) is not consistent:
7          ∃(s1,s2) ∈ S, a ∈ A, e ∈ E : row(s1) = row(s2) and T(s1·a·e) ≠ T(s2·a·e)
8          E ← a·e
9          extend T to (S∪S·A)·E using MQ
10     if (S,E,T) is not closed:
11         ∃s1 ∈ S, a ∈ A : ∀ row(s1·a) ≠ row(s)
12         S ← s1·a
13         extend T to (S∪S·A)·E using MQ
14   M = M(S,E,T)  # (S,E,T) is closed and consistent
15   if Teacher replies with a counter-example t:
16     add t and all its prefixes to S
17     extend T to (S∪S·A)·E using MQ
18 Halt and output M
```

Listing 1: The L* Algorithm

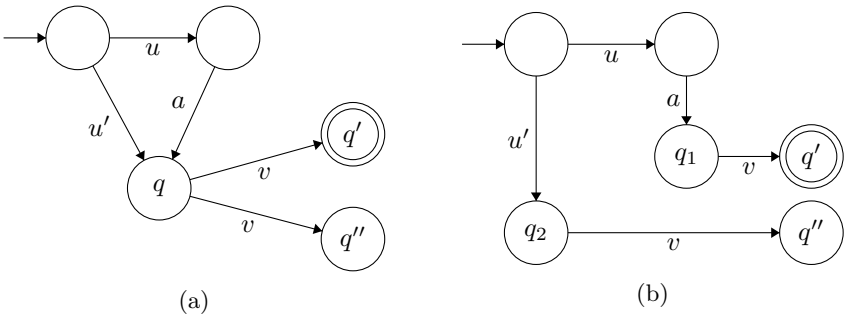

Figure 3: Formal progression of an incorrect conjecture: (a) inconsistent model for distinguishing suffix v from state q, (b) consistent model after splitting q into new states q_1 and q_2.

1.3 Counterexample Decomposition

In the previous section, the learner was tasked to infer the behavior of set $\mathcal{A}$ illustrated in Figure 1. At some point the learner inferred an incorrect conjecture H_0, depicted in Figure 2. The key step to improve H_0 towards $\mathcal{A}$ was to utilize a given counterexample word $w = ab$. This section discusses how the counterexample can be decomposed by the learner and elaborates on the process of how this decomposition leads towards the appearance of a new state.

In a general and abstract way, one can consider a counterexample word w which is a counterexample for an incorrect conjecture H. Word w then has a suffix av for which holds that any two access sequences $u, u' \in U$ reach the same state in H, we have: $\lambda(ua \cdot v) \neq \lambda(u' \cdot v)$. One could digest this even further by concluding that the state with access sequence u trailed with letter a is a different state in comparison to the state with access sequence ua. In other words $\lambda^A(\lfloor u \rfloor_H a \cdot v) \neq \lambda^A(\lfloor ua \rfloor_H \cdot v)$.

This situation is visually depicted in Figure 3a. A counterexample word w thus can also be decomposed in the three-tuple $w = \langle u, a, v \rangle$. Element v is called a distinguishing suffix, because it distinguishes the states $U[ua]$ and $U[u']$ from each other. Angluin's L* algorithm adds v to the set E, such that in the observation table the rows ua and u' differ from each other. This results in one more distinct row in the observation table, hence that state q is split into two states: one state q_1 that leads uav to q' and one state q_2 that leads $u'v$ to q''. The latter is depicted in Figure 3b.

In the running example of the former section the counterexample word $w = ab$ led to the appearance of a new state in the conjecture. Word w is a valid counterexample because $\lambda_H(ab) \neq \lambda_{\mathcal{A}}(ab)$. According to the decomposition, this is true because $\lambda_H(ua \cdot v) \neq \lambda_H(u' \cdot v)$ with the counterexample decomposition $w = \langle u, a, v \rangle = \langle \varepsilon, a, b \rangle$. In Figure 2 $H[ua \cdot v] = H[\varepsilon \cdot a \cdot b] = H[ab]$ is equal to the state $H[u'v] = H[\varepsilon \cdot b] = H[b]$, whilst $\lambda_A(ab) \neq \lambda_{\mathcal{A}}(b)$. The state $H[\varepsilon \cdot a] = H[a]$ should thus be added. This is achieved by splitting $H[\varepsilon]$, because both $H[\varepsilon]$ and $H[a]$ have the same output for λ. This results into two new states: $H[\varepsilon]$ and $H[a]$.

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

In *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning* [5] Isberner et al. recognizes that one of the computational expensive deprivations of Angluin's L* algorithm is a consequence of redundant entries in the observation table. The redundant entries are a result of suboptimal and possibly excessively long counterexamples provided by the MAT to the learner's equivalence queries. The prefix-part and the suffix-part of the counterexample contain elements that do not contribute to the discovery of new states even the distinguishing suffix possibly contain redundant elements. The L* algorithm can only establish a conjecture when the observation table is closed and consistent. As a result the learner also poses membership queries to fill out each new entry field that is created by the product of the counterexample's redundant prefix and suffix elements. To illustrate the presence of the redundant fields: in the observation table of the employed example dfa shown in Table 1c, the value 0 in the ε-column suffices to distinguish the $\mathcal{A}[b]$-state from the other two states. The remaining fields for these particular rows denoted by the b-suffix do not contribute to the distinctiveness of these rows, hence the suffix to distinguish the state $\mathcal{A}[b]$ from $\mathcal{A}[\varepsilon]$ and $\mathcal{A}[a]$ are redundant.

In order to overcome performance impacts caused by redundancies in counterexamples, Isberner et al. proposes the TTT algorithm. The TTT algorithm does so by utilization of a redundancy-free organization of observations in a discrimination tree. A discrimination tree adopts two sets $S, E \subset \Sigma^*$ that are non-empty and finite like Angluin's L* algorithm does, the only difference being that S consists of state access strings opposed to prefixes. Set E still contains distinguishing suffixes, which are referred to by Isberner as discriminators. The tree can then be modeled by a rooted binary tree where leafs are labeled with access strings in S and inner nodes are labeled with suffixes in E. The two children of an inner node correspond to labels $\ell \in \{\top, \bot\}$ that represent the λ-evaluation. Other studies apply different terminology, like $\ell \in \{+, -\}$, $\ell \in \{1, 0\}$ or adopt consistency in the direction their children have: a child to the left coincides with $\ell = \top$ and a child to the right coincides with $\ell = \bot$ [5, 6].

The discrimination tree is shaped in a way such that words can be $sifted$ through the tree. The process of sifting is administered to a word $w \in \Sigma^*$ that starts at the root of the tree. For every inner node $v \in E$ of the discrimination tree the sifting process passes one branches to the $\top$- or $\bot$- child depending on the value $\lambda(w \cdot v)$ until a leaf node is reached. The leaf node then indicates the resulting state with the corresponding access sequence for word w. The relation of the leaf node to its direct parent, either $\top$ or $\bot$, depict whether w is accepted or not. Each pair of states have then exactly one distinguishing suffix, which is the lowest common ancestor of the two leafs.

The TTT algorithm follows the following steps in order to infer a correct model:

1. Hypothesis Construction
2. Hypothesis Refinement
3. Hypothesis Stabilization
4. Discriminator Finalization

Hypothesis Construction

Following the example where dfa $\mathcal{A}$ from Figure 1 should be learned again, but now according to the TTT algorithm, the algorithm starts out with an initial hypothesis, which is the same hypothesis for every SUT. The initial state of the initial hypothesis $q_0^{H_0}$ is identified with the empty word $\varepsilon \in S$ and transitions are determined by sifting a state's access sequence $u \in S$ with elements $a \in \Sigma$, such that ua is sifted through the tree. For the initial hypothesis, this means that εa and εb are sifted in order to respectively determine the a and b transitions from the initial state $H[\varepsilon]$. Furthermore, a state $q \in Q^H$ is in F^H if the associated leaf is on the $\top$-side of the ε-node.

The initial state is the only state in the hypothesis, hence $S = \{\varepsilon\}$, and since q_0 is an accepting state, the q_0-leaf is the $\top$-child of the ε-root. The initial discrimination tree DT_0 thus looks like Figure 4a and the dfa H_0 corresponding to DT_0 is depicted in Figure 4b.

Hypothesis Refinement

Section 1.3 discussed that a counterexample word $w = \langle u, a, v\rangle$ must satisfy $\lambda^A(\lfloor u \rfloor_H a \cdot v) \neq \lambda^A(\lfloor ua \rfloor_H \cdot v)$. The output of the lambda-evaluation differs for $\mathcal{A}$ and H, because the words $\lfloor u \rfloor_H a$ and $\lfloor ua \rfloor_H$ lead to different states in $\mathcal{A}$, but the same state in H. This is based on their difference in output for v and $\mathcal{A}$ being canonical. The conjecture H can be corrected by introducing a new state transition from the state $H\lfloor ua \rfloor$ to a new state with access sequence $\lfloor u \rfloor_H a$.

A counterexample for H_0 could be $w = b$, because $\lambda^A(b) = 0 \neq \lambda^H(b) = 1$. The counterexample can also be read as $w = \varepsilon b \varepsilon$, thus the decomposition of the counterexample is: $u = \varepsilon, a = b$ and $v = \varepsilon$. Since state $H\lfloor ua \rfloor = H\lfloor \varepsilon b \rfloor = H\lfloor b \rfloor = q_0^{H_0}$ should be split to a new state $\lfloor u \rfloor_H a \to \lfloor \varepsilon \rfloor_H b$, a new state q_1 is introduced that can be reached by the transition $\delta(q_0, b)$. The outbound transitions from the new state are determined by sifting the access sequence of the new state with Σ. These adjustments are depicted in DT_1 and H_1 (Figure 4c and 4d).

Hypothesis Stabilization

Although the previous step of hypothesis refinement constructed a discrimination tree and a dfa conjecture that are consistent with each other, this does not necessarily need to be the case. There is a possibility that for a word that can be formed in $w \in S \times E$, a $\top$-child predicts the output 1 but the hypothesized conjecture 0 or a $\bot$-child predicts the output 0, but the hypothesized conjecture 1. Word w then forms a new counterexample, and is dealt with likewise the former counterexample. This step is done untill the hypothesis is stable and the discrimination tree and the conjecture are consistent.

Hypothesis Finalization

Suppose H_1 was subjected to an equivalence query, which resulted in the counterexample *abbbb*. Because $\lambda^A(\lfloor u \rfloor_H a \cdot v) \neq \lambda^A(\lfloor ua \rfloor_H \cdot v)$ only holds for $a = a$, the corresponding $\langle u, a, v\rangle$-decomposition of this counterexample thus is $\langle \varepsilon, a, bbbb\rangle$. Splitting $H\lfloor a \rfloor = q_0^{H_1}$ into a new state $\lfloor u \rfloor_H a \to \lfloor \varepsilon \rfloor_H a$ a new state q_2 is introduced that can be reached by the transition $\delta(q_0, a)$. The outbound transitions are established likewise the former counterexample.

The problem with the current counterexample, is that the discriminator $v = bbbb$ contains redundant information, whilst the entire discriminator will be added as a new inner node in the discrimination tree. For this reason, discriminators that are directly derived from counterexamples are treated as *temporary* discriminators. In Figure 4e this is shown with a dashed surrounding

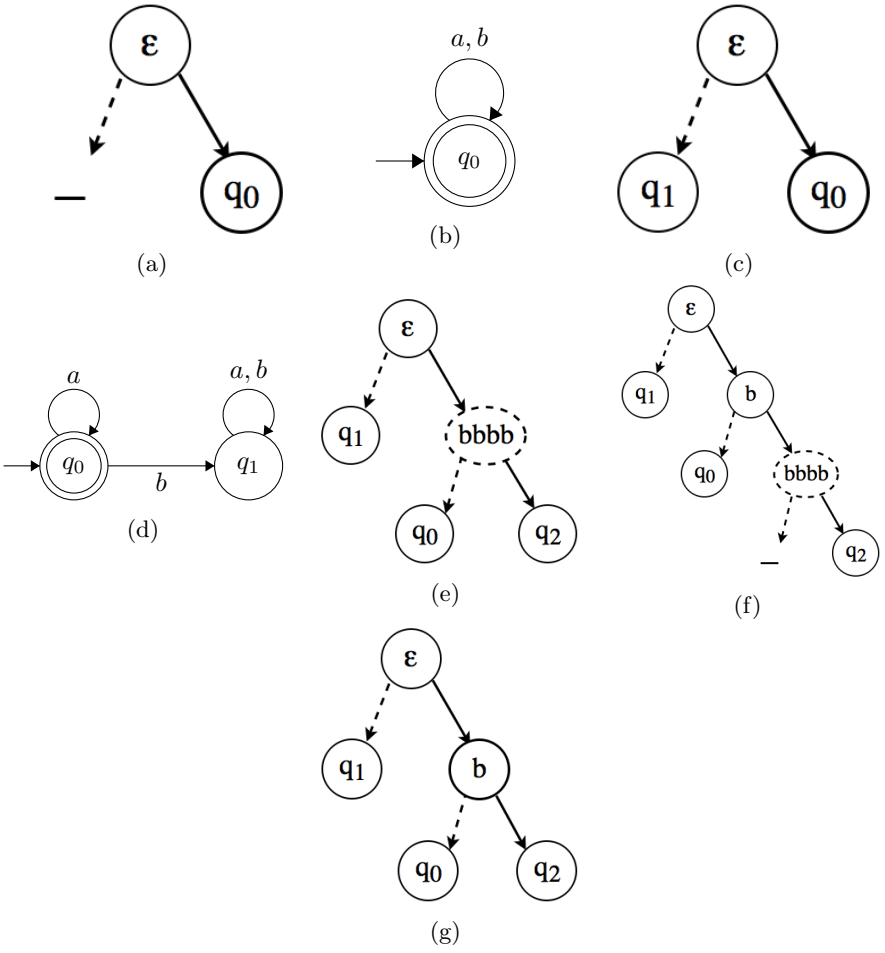

Figure 4: Evolution of discrimination trees and conjectures towards learning $\mathcal{A}$ with the TTT algorithm: (a): initial discrimination tree DT_0, (b) initial conjecture dfa H_0 corresponding to DT_0, (c) discrimination tree DT_1 after processing counterexample $w = b$, (d) the conjecture H_1 corresponding to DT_1. Evolution of the discrimination tree: (e) discrimination tree with *bbbb* as temporary node, (f) discrimination tree with b as discriminator, (g) final discrimination tree without temporary nodes.

of the inner node *bbbb*. Moreover, the subtree with the temporary discriminator as its root node is surrounded by a rectangular region, which is further referred to as a *block*. In order to remove the discriminator's redundancy, these blocks must be split by subsequential replacement of the temporary discriminator closest to the root by a final discriminator. New final discriminators v' are obtained by prepending a symbol $a \in \Sigma$ to an existing final discriminator or in other words adding an additional parent node in the block above the temporary node. This is shown in Figure 4f and 4g. Figure 4f shows the block with a root node and the temporary contents *bbbb*. Figure 4g shows the annex of a new root node in the block, parental to the temporary node.

References

[1] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[2] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

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

[4] E Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

[5] Malte Isberner, Falk Howar, and Bernhard Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In *RV*, pages 307–322, 2014.

[6] Therese Berg and Harald Raffelt. Model checking. *Model-Based Testing of Reactive Systems*, pages 77–84, 2005.