Bachelor's Thesis Computing Science

Radboud University Nijmegen

Optimizing $L^{\#}$ using optimal witness selection

Author:
Gijs Minkhorst
s1106692

First supervisor/assessor:
Prof. Dr. F.W. Vaandrager

Second assessor:
Dr. J.C. Rot

April 2, 2026

Abstract

We present an optimization of the active automata learning algorithm $L^{\#}$ by improving state identification. This reduces the number of learner queries needed for the identification of states, thereby reducing the total number of queries, by choosing informative witnesses. By applying our optimization to other parts of the algorithm we further improved the algorithm. Running an AALpy implementation on commonly used benchmarks suggests that the optimization leads to a significant reduction in the number of membership queries.

1 Introduction

You enter a room with a lamp, and there are two switches for turning off the lamp. You press one switch and the lamp stays on. So you walk to the other switch, press it, and the lamp still stays on. So you walk back to the first switch press it again and finally the lamp turns off. You conclude that the lamp only turns off if both switches are off, and when you entered the room the first switch was off and the second one was on. What you did is learn how the logic of the system works (both switches must be off) by giving inputs (pressing the switches) and getting output (the lamp is on or off).

Learning the workings of black-box systems is what active automata learning (AAL) is all about. Where systems can be, easy to comprehend real-life systems like lamp switches, a coffee machine or an ATM. It could be protocols like the SSH or TCP protocol, complete software systems or things like regular expressions. Basically everything that can be represented using automata (regular languages). A system like this will be called a SUL (System under learning).

In [Angluin, 1987] the L^* algorithm was presented which introduced a lot of the core concepts used in AAL. An AAL algorithm learns by using a learner and teacher method. The learner runs the learning algorithm. The learner can pose inputs to the teacher and get the corresponding output. The learner also can make hypothesis of what the unknown automaton could be, and then the teacher checks it, if it is not correct the teacher poses a counter example to the learning algorithm. This cycle will iterate until the learner hypothesizes the correct automaton.

The L^* algorithm makes use of an observation table to keep track of the received information from the teacher. Also, the algorithm focuses on equivalence of states to distinguish them. In [Vaandrager et al., 2022] the AAL algorithm $L^{\#}$ was introduced. In contrast to L^*, it uses apartness between states instead of equivalence. $L^{\#}$ also uses an observation tree instead of an observation table, a tree like automata, which better represents the struc-

ture of the hidden automaton.

Recently, a new algorithm called L^s was presented in [Schwabe et al., 2025]. It tries to take the best of observation tables and trees and introduces sparse observation tables. From this data structure it derives a granularity metric. Which is a good heuristic to improve state identification. The L^s algorithm performs better in some cases than the $L^\#$ algorithm.

In this paper we aim to optimize the $L^\#$ algorithm using the concept of the granularity metric as used by [Schwabe et al., 2025]. Then we are going to evaluate the improved version on the same benchmarks as the L^s algorithm.

2 Preliminaries

$L^\#$ is an automata learning algorithm like L^*. First we give context by elaborating on AAL. Then we show how the $L^\#$ algorithm works, and specify the most important parts. We have made use of the explanation used in [Kruger et al., 2023], a full example walkthrough can be found there as well. For exact definitions and workings of the algorithm we refer to [Vaandrager et al., 2022]. Following this we will explain relevant parts of L^s in order to explain the granularity metric from [Schwabe et al., 2025].

2.1 Active automata learning

Active Automata Learning (AAL) is about learning a black-box system which we called the System Under Learning (SUL). These are systems that can be represented using Finite State Machines (FSM). An FSM is a set of states that have transitions between them. These transitions are based on inputs. Depending on what kind of FSM we have different outputs. We will be discussing Mealy Machines, which have a specific output corresponding to a specific transition, so depending on current state and input. An example can be found in figure 1.

As mentioned before an AAL algorithm uses a learner and a teacher method. The learner can submit output queries (OQ) and equivalence queries (EQ) to the teacher. Output queries have a sequence of inputs to which the teacher responds with the corresponding outputs. Equivalence queries are hypothesis to which the teacher responds with YES if the hypothesis is correct or gives a counter-example if it is not correct. The teacher however does not know the inherent structure but can directly interact with the SUL using OQ's, so it is in-between the learner and the SUL. For OQ's from the learner the teacher can perform OQ's to the SUL. For EQ's received from the learner however it must get a counter example from the SUL by only using OQ's.

There are several strategies for the teacher to find counter examples, these methods are called equivalence oracles.

In practice AAL algorithms learn real systems, and one query to a system can take seconds since the teacher must reset the SUL to go back to the starting state. For this reason it is generally assumed that runtime of AAL algorithms are dominated by the SUL. So runtime is not a good metric to compare AAL algorithms. More generally they are compared based on queries and symbols

2.2 Workings of the $L^{\#}$ algorithm

The $L^{\#}$ algorithm focuses on apartness of states. We say states are apart if you have an input sequence which gives different outputs when started from the different states. Let states q, q' be apart we denote $q\#q'$. If the input sequence "σ" shows this apartness we denote $\sigma \vdash q\#q'$. This input sequence is called a witness.

For instance if we look at our lamp example again, let's say we have a room with a central lamp and in the room are two switches. If either switch or both are on, the lamp is on. The switches are a button, press once to turn the switch on, and twice to switch it off. In Figure 1 we have our finite state machine (FSM) that represents this system. Where we have the states $\{q_0, q_1, q_2, q_3\}$, the inputs $\{A, B\}$ which represents pushing switch A or switch B and the outputs {On, Off} based on whether the lamp turns on or off. Our starting state is q_0 which represent the state where both switches are off, and thus the lamp is also off.

In this FSM all states are apart. Take for example q_0 and q_2, if we take the witness $\sigma =$ "AB", and we do that input sequence σ from the state q_0, we get the output "On On" and if we do it from q_2 we get the output "On Off". This shows q_0 and q_2 are apart and $\sigma =$"AB" is a witness for them, so "AB" $\vdash q_0\#q_2$

For explaining the $L^{\#}$ algorithm the lamp system will be the SUL, the system which the algorithm must learn. It can query inputs to the teacher from which it gets the corresponding outputs. From these queries it can construct an observation tree to keep track of all the information. The states in our observation tree are divided in three different sets:

1. The basis states S which are pairwise apart (So really different states)

$$\forall p, q \in S, p \neq q : p\#q$$

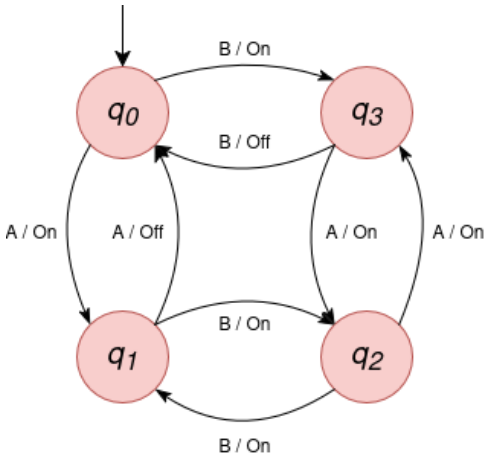


Figure 1: Mealy machine of Lamp system

2. The frontier states which are immediate successors of basis states but not in S

3. The other states which are not in the basis or in the frontier

For states in the frontier we have a Candidate Set which contains the basis states that could be the same as the state. So for a state r in the frontier the candidate set $C(r)$ is:

$$C(r) = \{q \in S | \neg r \# q\} \quad \text{So we have: } C(r) \subseteq S$$

Figure 2 shows an Observation tree in the process of learning our lamp system. We have the basis states $\{q_0, q_1, q_2\}$ in red, which are indeed apart, and the Frontier states $\{q_3, q_4, q_5, q_9\}$ in yellow. Each of the frontier states has a candidate set. Since q_9 has no outgoing transitions, it can not be apart from any state, and thus $C(q_9) = S$, which means q_9 could yet be any of the basis states. For the other states in the frontier we know that an input of "A" gives output "On", so the states are apart from q_2 but not from q_0 or q_1 So $C(q_3) = C(q_4) = C(q_5) = \{q_0, q_1\}$.

Now that we have our definitions clearly defined we can describe the actual $L^\#$ algorithm. The $L^\#$ algorithm consists of four rules: The promotion rule, which checks if we have found actual new states. The extension rule, which queries inputs for basis states. The identification rule, which tries to identify frontier states. And at last, we have the equivalence Rule, which checks whether we are finished. The rules are defined as followed:

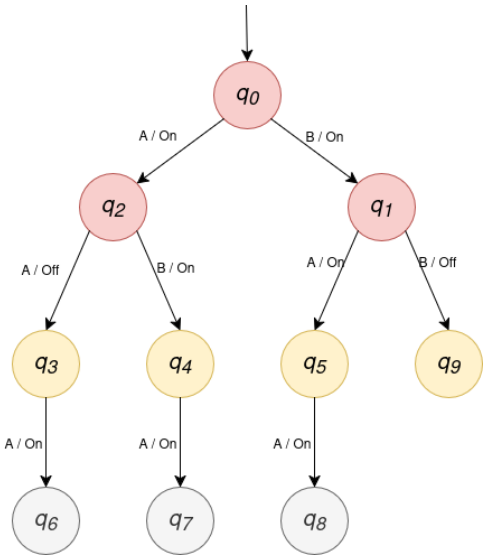


Figure 2: Observation tree

R1, Promotion Rule: If frontier F contains a state r with $C(r) = \emptyset$, then r is apart from all states in basis S, and therefore we may move it from F to S. If multiple states may be moved from the frontier to the basis then we may nondeterministically choose any of these states.

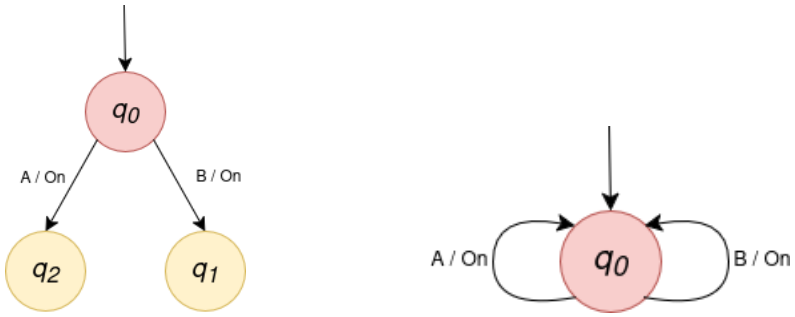
R2, Extension Rule: If some state $q \in S$ does not have an outgoing i-transition, for some input I, then the frontier is extended by adding a new state and an i-transition from q to this new state. The output of the new transition is determined via an output query access(q) i, where access(q) is the unique sequence of inputs leading to state q.

R3, Identification Rule: If frontier F contains a state r with a candidate set $C(r)$ that contains at least two elements, say q and q', then we pick a witness σ with $\sigma \vdash q \# q'$ and pose output query access(r) σ. In the resulting observation tree $q \notin C(r)$ or $q' \notin C(r)$.

R4, Equivalence Rule: If the first three rules cannot be applied, we construct a hypothesis from $\mathcal{T}$ and pose an equivalence query to the teacher. If the answer is YES then we are done, otherwise we process the received counterexample.

Now we have explained all necessary parts of $L^{\#}$, we will show a full walkthrough of $L^{\#}$ on our lamp system:

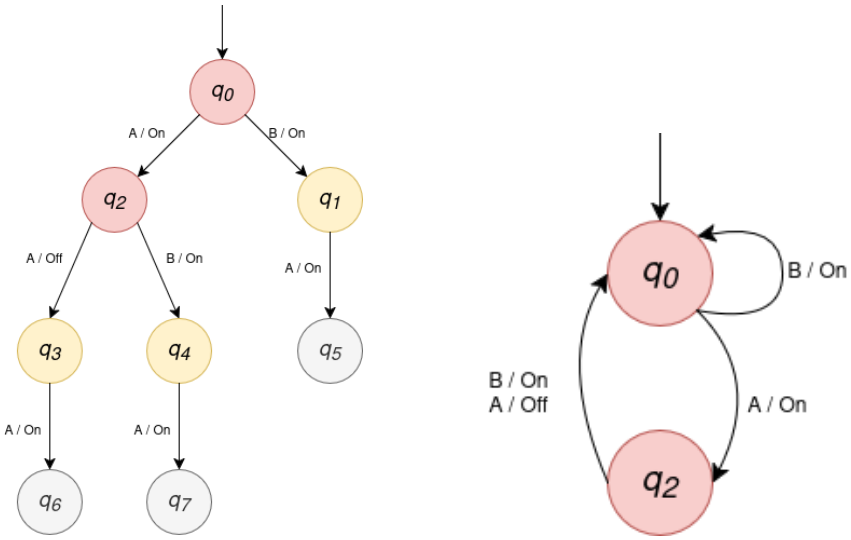
1. We start off with a single basis state q_0, since the basis consists of only one state and there are no frontier states, we apply the extension rule **R2** twice for both inputs, which gives us the two frontier states $q_1$and q_2. We get the observation tree shown in figure 3a. We cannot apply rule **R1**, **R2** or **R3**, so now we are going to apply **R4** and make a hypothesis. Because we cannot apply **R2**, we know that all basis states have outgoing transitions to frontier states. Also, because we cannot apply **R1** and **R3**, we know that for all frontier states q_f: $0 \neq |C(q_f)| < 2$ so $|C(q_f)| = 1$ which means there is only one unique candidate state for every frontier state. In our case we have $C(q_1) = C(q_2) = \{q_0\}$, so the transitions which in the observation tree go to q_1 and q_2, will go to q_0 in the hypothesis, as can be seen in figure 3b



(a) Observation tree (b) Hypothesis

Figure 3: First learning round

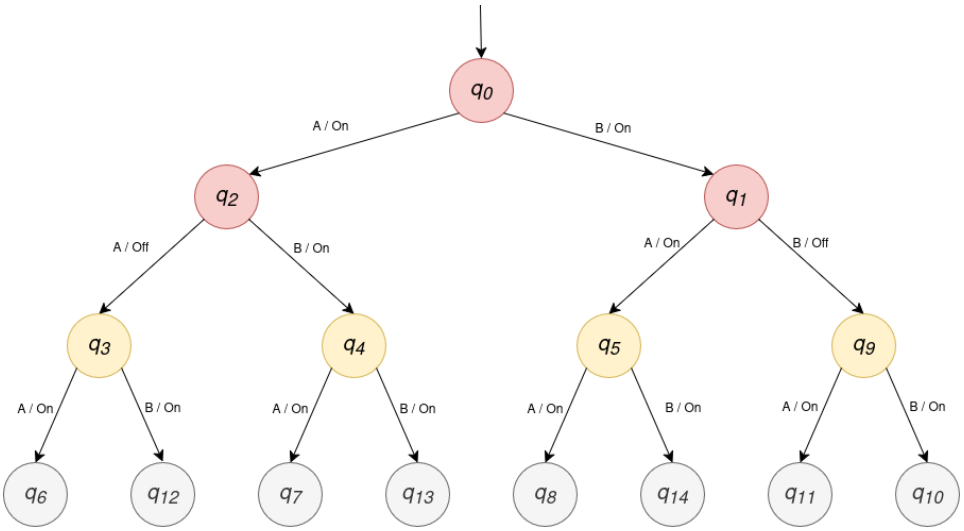
2. After proposing the hypothesis to the teacher with an equivalence query, we get the counterexample "AA", which gives the output "On Off". We add these transitions to our observation tree, and get new state q_3. Now we can apply **R1** to add q_2 to the set of basis states. q_3 is now a frontier state and **R2** is used to add the frontier state q_4. We now have three frontier states q_1, q_3, q_4, to which we apply **R3** three times, and get the new states q_5, q_6, q_7. Resulting in the observation tree of figure 4a. We cannot apply **R1**, **R2** or **R3** so we apply **R4**. Since $C(q_1) = C(q_3) = C(q_4) = \{q_0\}$ we get the hypothesis as shown in figure 4b



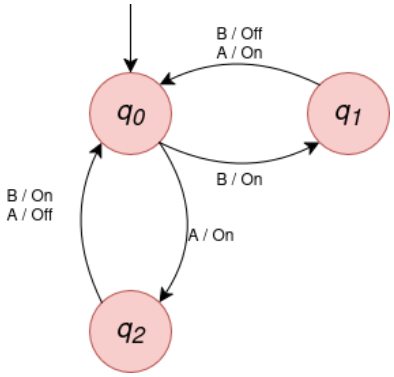
(a) Observation tree (b) Hypothesis

Figure 4: Second learning round

3. After proposing the hypothesis to the teacher, we get the counterexample "BAA", which gives the output "On On On". We add this observation to our observation tree which adds state q_8. We apply **R1** to add state q_1 to the basis. We promote q_5 to the frontier and use **R2** to add the frontier state q_9. We apply **R3** on q_9 which adds state q_{10}. We still have two states in the candidate set of q_9, so we apply **R3** again which adds state q_{11}. Since $C(q_3) = C(q_4) = C(q_5) = \{q_0, q_1\}$ we apply **R3** again on states q_3, q_4, q_5 which introduces states q_{12}, q_{13}, q_{14}. Now we have the observation tree as shown in figure 5a. We cannot apply **R1**, **R2** or **R3** so we apply **R4**. We have $C(q_3) = C(q_4) = C(q_5) = C(q_9) = \{q_0\}$ so we construct a hypothesis as shown in figure 5b



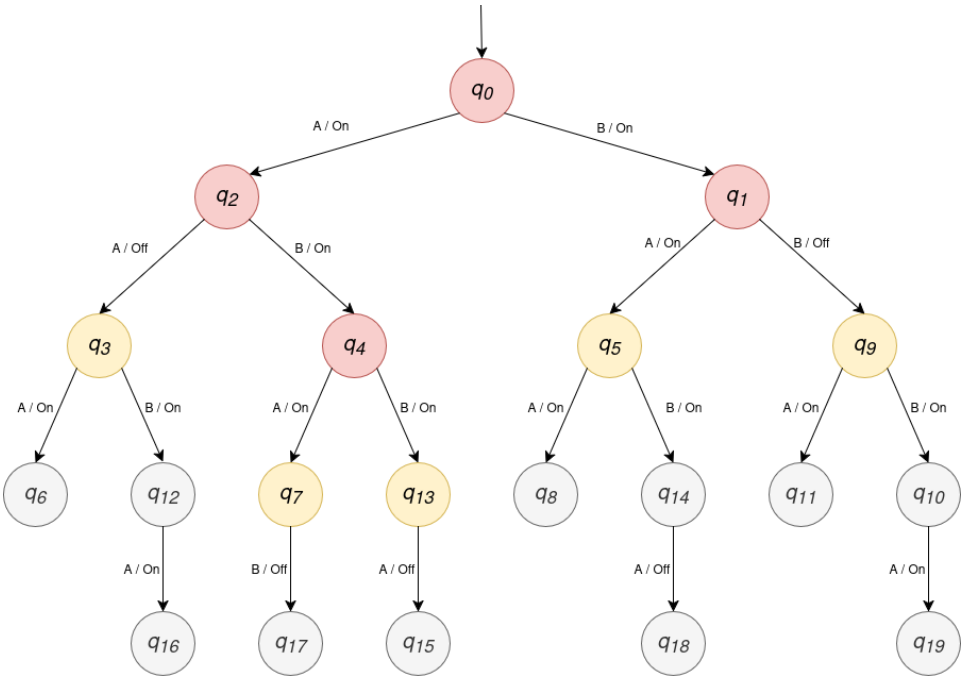
(a) Observation tree



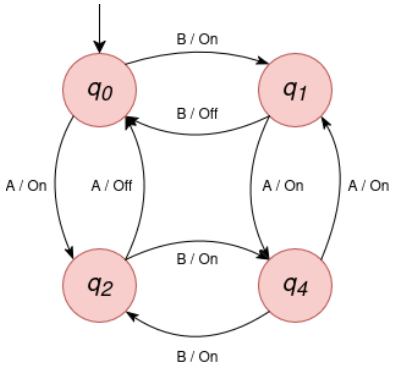
(b) Hypothesis

Figure 5: Third learning round

4. After proposing the hypothesis to the teacher, we get the counterexample "ABBA", which gives output "On On On Off", and adds the new state q_{15}. We apply **R1** to add q_4 to the basis and promote q_7 to the frontier. Then we apply **R3** four times on frontier states q_3, q_5, q_7, q_9, which adds the states $q_{16}, q_{17}, q_{18}, q_{19}$. Now we have our final observation tree, figure 6a, and construct the hypothesis, figure 6b. The hypothesis is queried to the teacher and the teacher responds with YES. As you can see the hypothesis is the same as the mealy machine of our lamp system as shown in figure 1. State names do not correspond, however, state names are arbitrary and thus do not matter.



(a) Observation tree



(b) Hypothesis

Figure 6: Fourth learning round

2.3 Granularity metric of L^s

Instead of an observation tree L^s uses a sparse observation table which is similar to an observation table. An observation table T has a set of prefixes U and suffixes V. Prefixes are input sequences done from the starting state and thus represents the target state reached by the prefix. Suffixes are input sequences used to distinguish between states represented by the prefixes, so they are similar to how witnesses work in $L^{\#}$. The set of prefixes consists of two disjoint sets $U = U_c \cup U_f$, $U_c \cap U_f = \emptyset$, which are the core prefixes U_c

and the fringe prefixes U_f. The core prefixes represent the confirmed states, where the fringe prefixes are the new states that need to be identified.

Table 1 is an example of an observation table for our lamp system. We have the prefixes: $U = \{\epsilon, A, B, AA, AB, BA, BB\}$ where $U_c = \{\epsilon, A, B\}$ and $U_f = \{AA, AB, BA, BB\}$. The suffixes are $V = \{B, BB\}$. The empty prefix is noted by ϵ, and τ corresponds to the last output of the prefix. Entries in the table are noted by $T(u, v)$, with $u \in U$ a prefix and $v \in V$ a suffix. For instance in our table we get $T(A, BB) =$ On On, which corresponds to our lamp system, if we have pressed A, then for pressing BB we get ouptut On On.

	τ	B	BB
ϵ	-	On	On Off
A	On	On	On On
B	On	Off	Off On
AA	Off	On	On Off
AB	On	On	On On
BA	On	On	On On
BB	Off	On	On Off

Table 1: Observation table

In an observation table fringe prefixes are identified by comparing the rows against the core prefixes. So for a fringe prefix $u_f \in U_f$ there is a unique $u_c \in U_c$ with $T(u_c, v) = T(u_f, v)$ for all $v \in V$. For example BB is identified with ϵ since $T(BB, B) = T(\epsilon, B) =$ On and $T(BB, BB) = T(\epsilon, BB) =$ On Off

The main difference of a sparse observation table is that it leaves the possibility for empty entries. If we look at table 2 we see a sparse observation table, it has some entries left. This is possible since we can still identify prefixes without knowing all entries, for example in table 2 we still can identify AB with A since $T(AB, BB) = T(A, BB) =$ On On. The difference that this makes is that the undefined $T(AB, B)$ does not have to queried, and thus the number of queries is reduced.

If we look in table 2 at the row of fringe prefix BB, we see we can not identify it at the moment. We want to find a suffix that bests identifies BB. This is where the granularity metric comes in to play.

Definition 7. *Let $U_r \subseteq U_c$ and $v \in V$. We define $D = \bigcup_{u_c \in U_R} \{T(u_c, v)\}$ as the set of distinct outputs produced by v across prefixes in U_r. The **granularity** of v with respect to U_r is* $\mathbf{gran}(v, U_r) = \max_{o \in D} |\{u_c \in U_r : T(u_c, v) = o\}|$ [Schwabe et al., 2025]

In our case the set of core prefixes we want to distinguish are all core prefixes so $U_r = U_c = \{\epsilon, A, B\}$. To choose a suffix to query we calculate the granularity of all suffixes with respect to U_r. For $v = B$ we get $D = \{\text{On}, \text{Off}\}$ which results in $\text{gran}(B, U_c) = \max\{|\{\epsilon, A\}|, |B|\} = \max\{2, 1\} = 2$. For $v = BB$ we get $D = \{$"On Off","On On", "Off On"$\}$ which results in $\text{gran}(BB, U_c) = \max\{|\{\epsilon\}|, |\{A\}|, |\{B\}|\} = \max\{1, 1, 1\} = 1$. We then choose the suffix with minimal granularity, which in our case is $v = BB$, we query $T(BB, v) = T(BB, BB) = \text{On Off}$, and we have identified BB with ϵ using one query.

So the granularity metric evaluates the effectiveness of suffixes in identifying fringe prefixes. If we are identifying u_f and it is compatible with U_r then the granularity metric $\text{gran}(v, U_r)$ says how many prefixes of U_r are at most compatible after querying $T(u_f, v)$

	τ	B	BB
ϵ	-	On	On Off
A	On	On	On On
B	On	Off	Off On
AA	Off	On	On Off
AB	On	-	On On
BA	On	On	On On
BB	Off	-	-

Table 2: Sparse Observation table

3 Research

Our goal is to improve the current $L^\#$ algorithm. These kinds of learning algorithms are typically evaluated on four different aspects: the queries and the symbols generated by the learner and the total number queries and symbols of the learner and teacher together. This paper will investigate the effectiveness of an optimization which focusses on reducing specifically the queries that the learner makes. The queries of the learner is a generally used measure for evaluating AAL algorithms. The queries and symbols of the teacher heavily depend on the chosen equivalence oracle and thus are a

less good measure then the learner queries and symbols. Under the assumption that SUL resets are the bottleneck for total runtime, and more symbols does not increase the number of resets, we conclude that queries are a better measure.

In the current version of the $L^{\#}$ algorithm, witnesses, which are sequences of actions to distinguish states, are chosen without any special reasoning. The metric-based suffix selection approach by [Schwabe et al., 2025] showed that by evaluating the effectiveness of identifiers you can choose your identifiers intelligently and reduces the number of queries. First we will explain our approach to this concept. Followed by a description and notes of our implementation and lastly our method of evaluating effectiveness.

3.1 Optimization

If we look at the Identification rule of the $L^{\#}$ algorithm we have a free choice in q and q' which are candid states for the state we want to identify. And the choice of our witness is also free.

In the L^{s} algorithm the granularity metric is used to choose a suffix which best reduces compatible core prefixes for a specific fringe prefix. In the $L^{\#}$ algorithm, we can see the compatible core prefixes as our candidate set, since they are states that are similar, and suffixes as witnesses since they distinguish states. So in our optimization we are going to choose a witness that reduces the number of states in the candidate set the most. Similar as the granularity metric we want to be able to assign a score to a witness based on how much states it can eliminate from the candidate set.

First we want only the witnesses σ for a candidate set of q:

$$W(q) = \{\sigma \in A^* | \exists r, r' \in C(q) : \sigma \vdash r \# r'\}$$

With A^* all input sequences in our observation tree
For all these witnesses we want to give a score to each witness based on how many states they show apartness of:

$$S(\sigma) = \frac{|\{(r, r') \in C(q) \times C(q) | \sigma \vdash r \# r'\}|}{2}$$

We divide by two because we count all the pairs twice, since $r \# r' \iff r' \# r$, in the implementation we only take half of the pairs since that is more logical and reduces the amount of work the algorithm does. From our witnesses W(q) we can now select the witness which gives the highest score, and thus distinguishes the most states from the candidate set and bests identifies state q.

$$\sigma_q = \max_{\sigma \in W(q)} S(\sigma)$$

So we have a different version for our Identification rule:

Identification Rule: If frontier F contains a state r with a candidate set $C(r)$ that contains at least two elements. Then we pick the optimal witness σ_r and pose output query access(r) σ_r. Then there is at least one state $q \in C(r)$ for which in the resulting observation tree $q \notin C(r)$.

To illustrate how this works we are going to look at Figure 2 again. We want to identify q_9 and we know that $C(q_9) = S = \{q_0, q_1, q_2\}$. In the original algorithm, we would choose 2 states from the Candidate set, for example q_0 and q_1 then we would pick a witness for example "B". Since querying "B" would give an output of "On" we know $q_1 \notin C(q_9)$ however still there are two states in the candidate set $C(q_9) = \{q_0, q_2\}$ Then we would need to apply rule 3 again with for example witness "A". Then we would get output "On" which implies $q_2 \notin C(q_9)$, so $C(q_9) = \{q_0\}$. This means we did 2 queries to identify the state q_9

In the new algorithm, we start with calculating all the witnesses:

$$W(q_9) = \{\sigma \in A^* | r, r' \in C(q_9) : \sigma \vdash r \# r'\} = \{"A", "B", "AA", "BA"\}$$

Then for each witness in $W(q_9)$ we are going to calculate the score:

$$S(\sigma) = \begin{cases} 2 & \sigma = "A" \\ 2 & \sigma = "B" \\ 3 & \sigma = "AA" \\ 2 & \sigma = "BA" \end{cases}$$

So then we take the witness with the highest score and we get σ_{q_9} ="AA". Querying σ_{q_9} gives output "On Off", which shows that $q_1, q_2 \notin C(q_9)$, and we get $C(q_9) = \{q_0\}$. So now we have identified q_9 using only one query.

3.2 Extending the optimization

Besides optimizing in the identification rule, we can also use our new method to improve the extension rule. In the AALpy implementation of $L^\#$ there is already a possibility to add a witness behind the new input. What that means is that when we extend our tree with a state, we try to identify it in the same query. However, this is also done by selecting just two states from the basis, and take just 'a' witness for them, but that may not be the best choice. We are going to apply the same optimization as in the previous section, however the state q we are going to add has no outgoing states yes so we get $C(q) = S$. The extension rule will then look like:

Extension Rule: If some state $q \in S$ does not have an outgoing i-transition, for some input I, then the frontier is extended by adding a new state and an i-transition from q to this new state. The output of the new transition is determined via an output query access(q) i σ_S, where access(q) is the unique sequence of inputs leading to state q, and σ_s is the optimal witness for the basis S

In the first learning round, when there is only 1 basis state, there are no witnesses (a witness must distinguish two basis states). However, when querying inputs we could add more inputs behind it. Instead, access(q) i we could do for instance access(q) i i or access(q) i i i i. We tested the improvements for both options. For the final version of our algorithm, if there is only one basis state, we query access(q) i i. We chose this based on tests results, which we will elaborate further on in section 4

3.3 Implementation

This optimization will be implemented in the AALpy tool[Muskardin et al., 2022]. AALpy is a python framework containing implementations for several AAL learning algorithms including $L^{\#}$. We chose this framework because it contains the most up-to date version of $L^{\#}$ and python is an accessible and intuitive programming language. All data and source-code of the implementation can be found online[1]

In algorithm 1 we describe an implementation of the described methods using pseudocode. We abstracted the code to be applicable for both witness selection optimizations. For the optimization of the identification rule mentioned in section 3.1 you would run this code on the Candidate set of your state you want to identify, so 'Basis_states $\leftarrow C(q)$'. For the optimization of the extension rule mentioned in 3.2 you would take the whole basis, so 'Basis_states $\leftarrow S$'. The implementation of the final optimization, that adds additional inputs in the first learning round, mentioned in 3.2, we consider trivial. The "Get_witnesses" function is implemented using a recursive depth first search of all possible input sequences, and checking if they show apartness. The "Combinations" function takes all unique combinations from a given set, which in python is a standard function.

3.4 Testing

To evaluate the effectiveness of the algorithm, we are going to test our implementation using various benchmarks, see table 3. These benchmarks are used to evaluate L^s in the paper from [Schwabe et al., 2025], and they

[1] https://gitlab.science.ru.nl/gminkhorst/bachelor-thesis

Algorithm 1 Pseudocode of Score-based witness selection

```
function SELECT_OPTIMAL_WITNESS(Basis_states)
    W = Get_witnesses(Basis_states)
    σ_opt ← None
    score_opt = −1
    for σ ∈ W do
        score ← 0
        for (q1, q2) in Combinations(Basis_states) do
            if σ ⊢ q1#q2 then
                score ← score +1
            end if
        end for
        if score > score_opt then
            σ_opt ← σ
        end if
    end for
    return σ_opt
end function
```

originate from the automata wiki [Neider et al., 2019]. The automata wiki contains various models for benchmarking AAL algorithms.

These benchmarks keep track of the number of queries and symbols the algorithm uses. As explained before we use the number of learner queries as our evaluation metric. Each of the benchmarks is run a 100 times, to account for randomness in the counter-examples given by the teacher. Of these results the median values are taken, since for the evaluation of L^s in [Schwabe et al., 2025] the same method is used. Similarly, we use the same equivalence oracle *RandomWp* with parameters $minSize = 0$ and $rndLen = 4$.

Most of the tests are run on an *Intel Core i5-6200U* processor. The larger benchmarks TCP and SSH take a lot of time in the optimized version. Since we run the benchmarks 100 times we made use of clusters from the Radboud University, this way we can run the benchmarks in parallel and save a lot of time. The benchmarks of TCP and SSH are run on an *AMD EPYC 7642* processor.

4 Results

In the previous sections we presented multiple optimizations to the $L^\#$ algorithm. In table 4 we can see the difference that each of these additions

Benchmark	States	Inputs
Passport	4	19
Volksbank_learnresult_MEASTRO_fix (Bankcard)	7	14
GnuTLS_3.3.8_client_full	15	12
Mosquitto_two_client_will_retain (MQTT)	18	9
TCP_server_ubuntu_trans	57	12
BitVise (SSH)	66	13

Table 3: Benchmarks

makes based on learner queries. $L^{\#}_{V_1}$ only uses the Identification rule optimization mentioned in section 3.1. The $L^{\#}_{V_2}$ version also uses the Extension rule optimization mentioned in section 3.2. The last two versions use both optimization of V_1 and V_2, and it uses the multiple inputs optimization done in the first learning round, also mentioned in section 3.2. $L^{\#}_{V_3}$ uses double input, and $L^{\#}_{V_4}$ quadruple input. Based on the results we take $L^{\#}_{V_3}$ as our final version since it uses all optimizations. The $L^{\#}_{V_4}$ algorithm did perform better, however not a significant amount, and the optimization is only for the first learning round and will thus not scale with bigger models.

Benchmark	$L^{\#}$	$L^{\#}_{V_1}$	$L^{\#}_{V_2}$	$L^{\#}_{V_3}$	$L^{\#}_{V_4}$
Passport	157	142	138	135	135
Bankcard	224	221	199	193	193
TLS	205	207	223	208	205
MQTT	431	415	385	381	379

Table 4: Learner queries, all versions

In table 5 we present all results of the $L^{\#}_{V_3}$ version. Since the bigger benchmarks (TCP and SSH) take a long time to run, we only ran them for our final version. In comparison, L^s can learn a printer model in no longer than a minute. The printer model has 3410 states and is significantly larger than the benchmarks of TCP and SSH which have around 60 states which takes $L^{\#}_{V_3}$ several minutes to learn. On the printer model the optimized version of L^s has a performance of $30\mu s$ per query. For the benchmarks TCP and SSH the $L^{\#}_{V_3}$ algorithm needed around $10ms$ per query.

In table 6 and figure 7 we compare the number learner queries of $L^{\#}_{V_3}$ against

Benchmark	LO queries	LO sym	Overall queries	Overall sym	time (sec)
Passport	135	552	290	1475	∼0
Bankcard	193	999	265	1479	∼0
TLS	208	1153	214534	1638698	8
MQTT	381	2902	882	6580	3
TCP	2591	30766	54740	634425	1026
SSH	2422	30385	223183	2563957	895

Table 5: All results of $L^{\#}_{V_3}$ (Median values, 100 runs)

the original $L^{\#}$ algorithm, and the L^s algorithm from [Schwabe et al., 2025]. We also have included $L^{\#}$-ADS. $L^{\#}$-ADS is an adaptive version of $L^{\#}$. Adaptive AAL algorithms are a little bit different from the earlier discussed algorithms. What makes it adaptive is that it can choose next inputs of a query on the fly. In other words it can do the inputs of a query step-by-step and depending on the outputs it can apply different inputs. One adaptive query however counts as one query even tough you can perform better than a normal AAL algorithm. Even tough $L^{\#}$-ADS has this advantage, the new $L^{\#}_{V_3}$ comes really close to the number of queries of $L^{\#}$-ADS

For the most benchmarks (Passport, MQTT, TCP, SSH) $L^{\#}_{V_3}$ performs better than $L^{\#}$ and L^s, and it comes close to the performance of $L^{\#}$-ADS. The Bankcard benchmark is the only benchmark where L^s outperforms $L^{\#}_{V_3}$, but it also outperforms $L^{\#}$-ADS. An explanation could be that the total different approach of the L^s algorithm is just a little bit better fit for this specific benchmark. The TLS benchmark also gives unexpected results, which also can be clearly seen in table 4, for this benchmark the optimizations don't affect or worsen the performance of $L^{\#}$. Choosing optimal witnesses at a specific moment is a logical decision to reduce learner queries, which is reflected in most results. A possible reason for the increase of learner queries is that the new algorithm chooses the shortest optimal witness, whereas the original algorithm may have coincidently chosen longer optimal witnesses. Although both use optimal witness and identify states just as well, longer witnesses give more information and can possibly reduce queries in later cycles of the algorithm.

Benchmark	$L^{\#}$	$L^{\#}_{V_3}$	L^{s}	$L^{\#}$-ADS
Passport	157	135	153	135
Bankcard	224	193	184	194
TLS	205	208	270	249
MQTT	431	381	432	375
TCP	3004	2591	2618	2512
SSH	3054	2422	2789	2407

Table 6: Learner queries for different algorithms

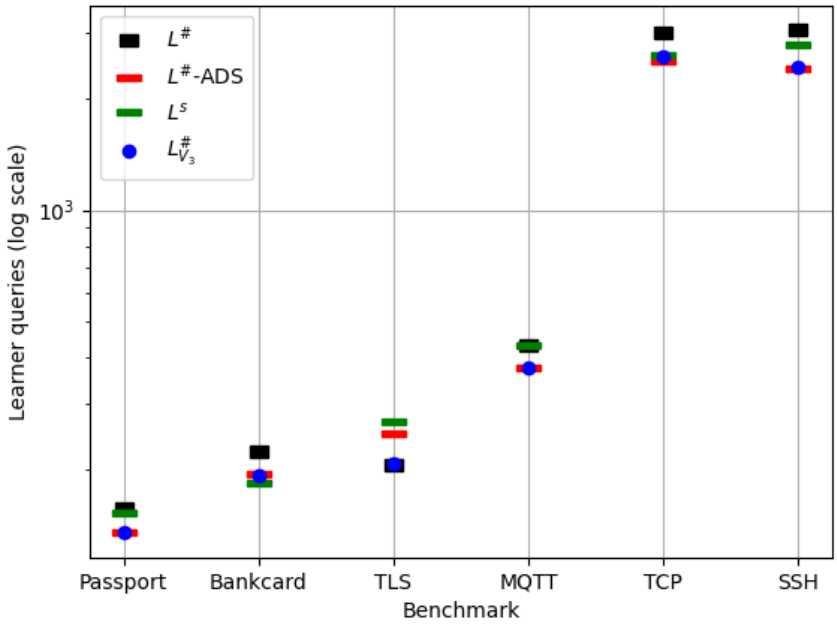


Figure 7: Performance of different algorithms

5 Conclusion

The goal of this research was to improve $L^{\#}$ using ideas from [Schwabe et al., 2025]. The granularity metric of L^{s} evaluates effectiveness of suffixes in state identification. We similarly developed a score function for witnesses, which evaluates how many candidate states the witness can distinguish. Using this score we can select better witnesses in the extending and identification phase of the $L^{\#}$ algorithm. This results in having to query fewer witnesses, and reduces the number of learner queries. Testing an implementation on

the same benchmarks as L^s shows it performs way better then $L^\#$, and most of the time better than L^s. The $L^\#_{V_3}$ even performs almost just as well as the adaptive $L^\#$-ADS algorithm. So we can conclude a significant improvement. However, L^s is still better scalable than $L^\#_{V_3}$ because of runtime performance.

6 Future Work

There are multiple aspects of the improved $L^\#_{V_3}$ which could be further investigated or improved. As mentioned in the section 4, the results of the Bankcard benchmark show there are still some cases where L^s performs better. A thorough inspection of both algorithms could possibly reveal areas of improvement. Also, the TLS benchmark showed that the $L^\#$ algorithm can perform better than $L^\#_{V_3}$. Does this mean $L^\#$ randomly chooses better witness then $L^\#_{V_3}$, and should our metric of good witnesses be changed, or is there something else going on? Since from this small set of benchmarks there are still some surprising results, it would be valuable to test $L^\#_{V_3}$ on a bigger collection of benchmarks. There could be more cases where $L^\#_{V_3}$ does not perform the best. Furthermore, while learner queries are typically considered the best performance metric, the learning algorithm does influence the testing queries. This suggests a potential trade of where a decrease of learner queries in $L^\#_{V_3}$ increases testing queries. It is important to take this into account and future work could use alternative or combined performance metrics. Lastly, we note that the runtime of the algorithm is greatly increased for big models. This is not a surprising outcome because this research focuses on optimal queries. As we stated in section 2.1 the runtime of an AAL algorithm is not really an issue since the in practice the SUL dominates runtime. However, that does not mean the algorithm should be unnecessarily slow. There are a lot of different ways in which the program can be optimized runtime wise. In our new algorithm, for every state in the frontier, are all witnesses and witness scores calculated, for every time the extension or identification rule is applied. We could only do it sometimes, or we could keep track of all witnesses and their scores and update them accordingly every time, this would save a lot of computation time.

References

[Angluin, 1987] Angluin, D. (1987). Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75 (2), pages 87–106.

[Kruger et al., 2023] Kruger, L., Garhewal, B., and Vaandrager, F. (2023). Lower bounds for active automata learning. In *Proceedings of 16th edition*

of the International Conference on Grammatical Inference, volume 217 of *Proceedings of Machine Learning Research*, pages 157–180. PMLR.

[Muskardin et al., 2022] Muskardin, E., Aichernig, B. K., Pill, I., Pferscher, A., and Tappler, M. (2022). Aalpy: an active automata learning library. *Innov. Syst. Softw. Eng.*, 18(3):417–426.

[Neider et al., 2019] Neider, D., Smetsers, R., Vaandrager, F., and Kuppens, H. (2019). Benchmarks for automata learning and conformance testing. In *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, pages 390–416. Springer International Publishing.

[Schwabe et al., 2025] Schwabe, W., Kogel, P., and Glesner, S. (2025). Learning mealy machines with sparse observation tables. In *Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems*, pages 176–194. Springer International Publishing.

[Vaandrager et al., 2022] Vaandrager, F., Garhewal, B., Rot, J., and Wißmann, T. (2022). A new approach for active automata learning based on apartness. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–243. Springer International Publishing.