Master Thesis
Software Science

Radboud University

Model Learning Performance When SUL Inputs Have Different Costs

Author:
Johan Sijtsma
s4793676

Supervisor/assessor:
Dr J. S. L. Junges
`sebastian.junges@cs.ru.nl`

Second assessor:
Dr J. C. Rot
`jurriaan.rot@cs.ru.nl`

Advisor:
L. Kruger
`loes.kruger@cs.ru.nl`

September 10, 2024

Contents

1 Introduction

Model Learning, or Automata Learning, can be used to create a formal model that describes the behaviour of a computer program or a hardware device. The model of this so called System Under Learning (SUL) could, for example, be used to find implementation errors. With model learning techniques it is possible create a model that describes the behaviour of the SUL without any prior knowledge of its internal workings[23], making it especially useful for systems with no or scarce documentation. In Active Automata Learning we learn this behavioural model by actively giving the SUL inputs and observing its output as a reaction to those inputs. Typically, we know the inputs and outputs of the SUL before starting the learning process. For example, say we are trying to learn a behavioural model of a server. Some of the inputs could then be the echo command, or an order to reboot the server. Fitting outputs could then be that same message back for the echo command, or a start-up message for the reboot. With enough observations we can create a formal model, even without knowing how this server works on the inside.

Experimentally finding the behaviour of a system through inputs and observation of outputs was already shown to be exponential for finite automata by Moore in 1956[16]. Angluin introduced a way to learn finite automata in polynomial time in 1987[1] when an upper limit on the number of states is known. The broad strokes of Angluin's algorithm are still used in modern model learning algorithms, now called the MAT (Minimally Adequate Teacher) Framework.

Angluin sees the learning process as a game between a learner and teacher. The teacher has perfect knowledge of an SUL. The learner only knows the inputs and outputs. The learner's goal is to recreate a model if the SUL. The moves the learner can make in this game are as follows:

- An Output Query (OQ) in which the learner sends the SUL an input sequence. The SUL then responds with an output sequence.

- An Equivalence Query (EQ) in which the learner presents a hypothesised model to the teacher. If this hypothesis does not match the behaviour of the SUL, then the teacher gives the learner a counterexample, a sequence where the output between the hypothesis and the state diagram is different.

The Minimal in MAT stands for the assumption that the teacher gives good counterexamples from which the learner can learn more than from other counterexamples. The idea being that you need less counterexamples if they are helpful.

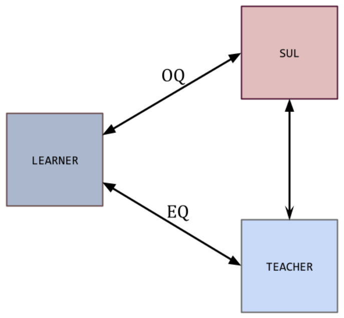


Figure 1: MAT Framework diagram.

Peled et al. [17] and Groce et al. [9] describe a way to apply such algorithms to software and hardware systems. We can perform an OQ by first resetting the SUL to an initial state and then observing the output sequence in response to the input sequence.

The EQ can be simulated as well, but takes a bit more effort. We do not always have access to a teacher that instantly knows wether the given hypothesis is correct, or that instantly knows a counterexample. Instead we compare the output of the SUL with the output of the hypothesis. We give the SUL many input sequences until the SUL produces an output sequence that is different from the output sequence the hypothesis predicts. If no input sequence is a counterexample, then the hypothesis is correct and the learning process is finished.

However, this process can take many input sequences. Worse still, once you have verified that no counterexample exists of length n or below, you still do not know that no $n+1$ length counterexample exists. As such it is not possible to find a finite set of input sequences to fully simulate an EQ. However, if we know that a SUL has at most k states, then we can construct a finite set of input sequences that contains at least one counterexample[12]. I will discuss some methods to create such a set in Section 2.4 on counterexample generation.

The performance of model learning techniques can be measured by using the number of inputs[8, 6] or number of queries [22, 20] used to learn the SUL. However, not all SULs have inputs or queries that should be weighted equally. SULs could have some inputs that are, compared to other inputs, more expensive take longer to process. If we look at the server example again, a reboot would take much longer than an echo. For learning SULs like this, it could save time or costs by using model learning techniques that use as few expensive inputs as possible.

Outline

For this thesis I implemented various measures that try to reduce the cost of learning a model when the cost of each input is known. Section 3.2 defines the Mealy Machine with weighted input. This input weight can be used to calculate the cost an input sequence and also how 'expensive' the full learning process was. This enables us to measure which model learning techniques are cheaper. In Section 3.3 various ways of decreasing the incurred cost during the EQ are described. The implementation of these measures can be found in this GitLab repository.
This thesis will research whether these measures can make learning a model cheaper, or make finding a counterexample cheaper. This will be tested in Section 4. Here we compare the L# model algorithm without any measure to L# with cost saving measures. In these experiments we learn models where a few inputs will be given a higher cost than the rest. Additionally we will find whether finding a single counterexample given a hypothesis of a model is cheaper with the measures than without the measures.
Finally, Section 5 summarizes the results for each and gives a recommendation for which measures to use in which situation.

2 Background Information

This section contains a brief overview of required knowledge for this paper, as well as related research. Readers familiar with concepts described in this section can use it as a refresher, or skip it entirely.

2.1 Mealy Machine

Model Learning revolves around the Mealy Machine. In this section I define what a Mealy Machine is as well as other related concepts that will be relevant in the context of this thesis.

A Mealy Machine is a type of state diagram. The Mealy Machine is used to describe the behaviour of the SUL. An example can be seen in Figure 2. At first the system is in state s_0, where you can perform inputs i_1 or i_2. If we use input i_1, we get output o_1 and move to state s_1, with i_2 we get output o_2 and stay in state s_0.

This definition is based on the definition from *Model Learning* by Vaandrager [23].

Definition 2.1. *A Mealy Machine is the tuple $(S, s_0, I, O, \delta, \lambda)$, where S is a set of states, O is a set of all outputs and I is the set of all inputs. $\delta : S \times I \to S$ is a transition function, while $\lambda : S \times I \to O$ is the output function. $s_0 \in S$ is the initial state.*

This thesis mostly uses finite complete Mealy Machines, so we assume that sets S, I and O are finite and functions δ and λ are defined for all elements of S, I and O unless otherwise specified.

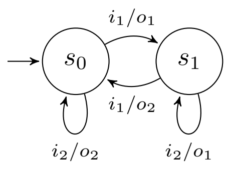


Figure 2: A visual representation of a simple Mealy Machine

It is often useful to consider multiple inputs at once. Definition 2.2 does exactly that. With this definition we can use an input sequence into the output function to get an output sequence in return.

Definition 2.2. *An input sequence is a word $\sigma \in I^*$. We overload output function λ for input sequences as follows: for all $i \in I, s \in S, \sigma \in I^*$, $\lambda(s, \epsilon) = \epsilon$ and $\lambda(s, i\sigma) = \lambda(s, i)\lambda(\delta(s, i), \sigma)$.*

Definition 2.3 defines ways to talk about Mealy Machines and input sequences. If two Mealy Machines respond with the same output sequences for all input sequences, then they are *equivalent*. This distinguishing sequence is most often used to show that a learned model is not equivalent to the target model. When this sequence starts from the initial state, then the sequence is called a *counterexample*.

Definition 2.3. *Mealy Machines M and N are equivalent ($M \approx N$) when $\lambda_N(s^0_N, \sigma) = \lambda_M(s^0_M, \sigma)$ for all $\sigma \in I^*$, where s^0_N and s^0_M are the initial states and λ_N and λ_M are the output functions for N and M respectively.*

A sequence $\sigma \in I^*$ is said to distinguish M and M iff $\lambda(s^0_N, \sigma) \neq \lambda(s^0_M, \sigma)$

For model learning, it is important for the Mealy Machine that you are learning to have a way to return to the same state (usually the initial state). We represent this with a separate 'reset' symbol that resets the Mealy Machine to the initial state.

Definition 2.4. *A Mealy Machine with reset is the tuple $(S, s_0, I \cup \{\textbf{reset}\}, O, \delta, \lambda)$, where S is a set of states, O is a set of all outputs and $I \cup \{\textbf{reset}\}$ is the set of all inputs with a separate reset symbol. $\delta : S \times I \cup \{\textbf{reset}\} \to S$ is a transition function, while $\lambda : S \times I \to O$ is the output function. $s_0 \in S$ is the initial state. For every $s \in S, \delta(s, \textbf{reset}) = s_0$*

Whenever a 'Mealy Machine' is mentioned in this thesis, you could also read 'Mealy Machine with reset', unless specified otherwise.

2.2 L*

The L* algorithm was originally published by D. Angluin in 1987[1]. As described in the introduction, a learner uses Output and Equivalence Queries to learn the SUL. First, the learner fills up an observation table using OQs. Then the learner then makes a hypothesis based on this observation table and makes an EQ to the teacher. The teacher either answers that the hypothesis is correct, or gives a *counterexample*. This counterexample is added to the observation table. The learner then fills up the table based on the counterexample with OQs again. This process repeats until the teacher is satisfied.

2.3 L#

L# (L-Sharp) is a learning algorithm that makes use of the apartness operator #[24]. The expression '$p\#q$' means that states p and q are *apart* from each other. L#, just like L*, makes use of Output and Equivalence Queries, but does not use an observation table, but a tree structure instead. This tree shaped partial automaton records all observations during the learning process. An example of such an observation tree can be seen in Figure 3. The #-operator is used while building the hypothesis. For example, input sequence i_1 from state p and q produces two different output sequences (o_2 and o_1 respectively). We can say that that i_1 is the *witness* to $p\#q$. If a witness exists between two states of the observation tree, then those must be two distinct states in the hypothesis. With this knowledge we can build the hypothesis, which can then be verified with an Equivalence Query.

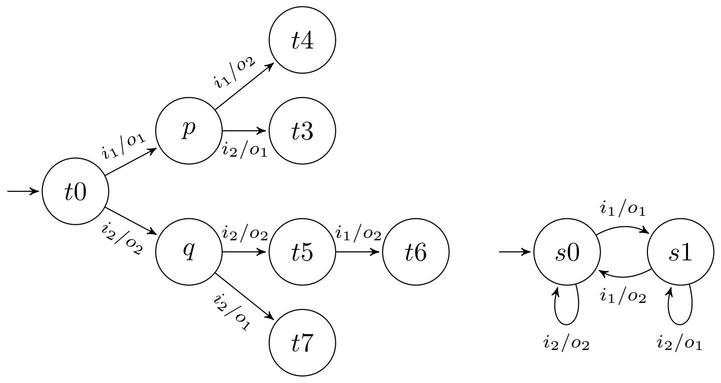


Figure 3: An observation tree (left) for a Mealy machine (right).

2.4 Counterexample Generation

As discussed in the introduction, to automate the EQ, you need a way to find a counterexample to the current hypothesis programmatically. In essence this means finding a set of input sequences of which at least one shows that the hypothesis is different from the SUL. This is very similar to the problem that conformance testing tries to tackle. In conformance testing we test whether a model conforms to its specification model. In model learning, we can use these conformance testing techniques to create a set of input sequences to programmatically discover whether our hypothesis 'conforms' to our SUL. A few methods of creating this test suite that are relevant to the rest of this thesis are briefly covered below.

All methods below assume that the model of the SUL will have at most n states, then we can construct a finite set that contains a counterexample. This was proven by both Chow[5] and Vasilevskii[25], who also worked on the first of the discussed method.

2.4.1 W-Method

The first method to create this test suite is the W-method. W is the name of the characterisation set. A way to make a minimal characterisation set was published by Vasilevskii[25]. Chow presents a recipe to construct this set of test sequences[5], but the following descriptions is based on the description found in Moerman's *Nominal Techniques and Black Box Testing for Automata Learning*[15].

The W-method test suite set is defined as follows:

Definition 2.5. $T_W = (P \cup Q) \cdot I^{\leq k} \cdot W$

An input-sequence from the W-method test suite consists of three parts. First, it starts with a section from either the P or Z set. P is the set of all access-sequences. An access-sequence for state s is an input-sequence that moves you from the initial state to state s. Q is the set $P \cdot I$, covering all possible transitions within the model. The next part of the input sequence is from set I^*, but only up to a certain length. We call the i from I^* the random walk. The maximum length of this walk can be defined with k. The final part is a separating sequence from the set W. A separating sequence w_{ab} from this set *distinguishes* one state (a) from one other state (b). The W set contains at least a separating sequence for all pairs of states.

In this thesis I will use P, I and W to refer to the sets and p, i and w, to refer to their elements.

Once these parts are concatenated, they are part of the test suite set. Using this set we can guarantee at least one counterexample, if one exists.

This method later was further improved in the Wp-method by Fujiwara et al. [7]. In the Wp-method, we first see which state was reached by the first part of the test sequence $(P \cdot I$ or $Q \cdot I)$. Call this state a. We then only concatenate separating sequences $w \in W$ where w distinguishing that state a.

2.4.2 Hybrid-ADS

The Wp-method was improved upon by the HSI-method by Luo et al. and Petrenko et al. [14][18]. Here, H contains separating sequences for every state:

Definition 2.6. $T_{Wp} = (P \cup Q) \cdot I^{\leq k} \cdot H$

The ADS-method[13] uses Adaptive *Distinguishing* Sequences (in set Z) instead:

Definition 2.7. $T_{ADS} = (P \cup Q) \cdot I^{\leq k} \cdot Z$

A distinguishing sequence is a separating sequence that can distinguishes all states from all other states. However, not every model has a distinguishing sequence, for example when two different state-pairs need two different inputs at the start of the seperating sequence for you to be able to tell them apart.

With an Adaptive Distinguishing Sequence, you instead have interact with the SUL one input at a time. Depending on the output you can by you change the next input symbol. In essence, the ADS takes the form of a tree, splitting on the output.

Hybrid-ADS combines elements of the ADS and HSI methods so it can be used in all situations[21]. Specifically, it start with ADS and fills in the gaps with HSI style sequences.

2.4.3 Randomized

Another method is to not worry about the finite nature of the Mealy Machine and just generate a single random input-sequence at a time[15]. The way this is done in the L# learning library (and the rest of this thesis) is by taking a random access-sequence (from P), a random walk of random length (from I) and a random distinguishing sequence for the state reached by the access-sequence and random-walk (from W). While this method has less theoretical guarantees, it works well in practice and does not require much memory when creating test suite for large models.

2.5 Other research

Other ways to reduce learning costs have been researched as well. For example, by removing or limiting the amount of resets needed to learn a SUL. Often times it is not the inputs, but the reset that is expensive, e.g. resetting a server. In such cases it is better to use as few resets as possible. Bremond and Groz note Hw-Inference as a possible way to learn without resetting [4]. Rivest and Schapire presents an algorithm that can infer a finite automaton using homing sequences [19]. A Homing sequence is an input sequence where, depending on the output sequence, we can determine at which state the SUL is. However, this technique can only be used for fully connected models. Models where you can every state can be reached from every other state.

3 Methods

This section first discusses how current model learning algorithms could perform better when optimising for input cost. We define a Mealy Machine with weighted input, as well as functions to calculate the cost of an input sequence. With these functions we can calculate how expensive it was to learn a model. Afterwards, possible avenues for reducing expenses are noted including ones that were implemented for this thesis. Then these implemented avenues are discussed in greater detail. Finally, we show how we set up experiments to see how the solutions performed.

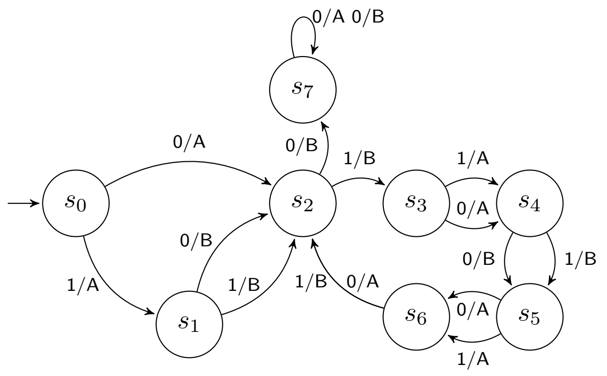


Figure 4: Example of a Mealy Machine where taking weighted inputs into account could be beneficial.

3.1 Problem Statement

When evaluating the performance of a model learning algorithm, the number of queries (OQs and EQs) is used[22, 20]. This is because the performance bottleneck for learning algorithms is generally the communication with the SUL within the OQ and EQ. Thus, reducing the amount of queries, or number of inputs within those queries, is a good way to improve performance.

However, not all inputs are created equal. Cases exist were some inputs are vastly more time-consuming than others. Resetting the SUL can take a long time too. For example, starting or rebooting a server can take minutes, while a simple echo takes milliseconds. When learning a model of this server it might be more effective to minimise the amount of server restarts at the expense of a higher total inputs during the learning process. Similarly, if the inputs cost money instead of time, minimising the amount of expensive inputs could result in a cheaper model learning process. For cases like these, a cheaper or faster way to learn the model might exist, but current model learning algorithms will not and have no way of finding the cheaper or faster route to learning their target system.

To solve this issue, I implemented a way to measure the cost (rather than the amount of inputs) of the learning process on an existing model learning library. Afterwards I implemented various measures to reduce this cost. Finally, I did experiments and report how efficient each measure is.

I implemented these measures on top of the L# Learning Library[11]. I chose the L# library because I could easily get help from the people working on L# at Radboud University. I also wanted to gain experience writing code in Rust. The implementation can be found in this GitLab repository, as well as some scripts I used to run the experiments. The specifics of the implementation are not that important for this thesis. The measures as described in the following sections could be implemented in any language on any model learning algorithm.

3.2 Definition of a Mealy Machine with Weighted Iinputs

To help with reducing cost in cases where inputs have different costs, we need a way to measure that cost. To do this we define a Mealy Machine where each input has a weight:

Definition 3.1. *The new input is a Mealy Machine* $(S, s_0, I \cup \{\textbf{reset}\}, O, \delta, \lambda, C, c_R)$ *as defined in Definition 2.1, with addition of the cost function* $C : I \cup \{\textbf{reset}\} \to \mathbb{N}$*, where* $\mathbb{N}$ *is the set of natural numbers.* $c_R \in \mathbb{N}$ *is the reset cost.*

If we take the Mealy Machine in Figure 4 as an example. We could define C(0) as 8 and C(1) as 2. From s_0 we could either go to state s_2 for the cost of 8 with input 0, or state s_1 for the cost of 2 with input 1. We can then use input 1 again to get to state s_2 for the lower cost of 4 than if we had only used input 0.

Similarly to the output function, we want to use whole input sequences within this function:

Definition 3.2. *We overload the cost function* C *for input sequences* σ*. For all* $\sigma \in I^*, n \in I$*:* $C(n\sigma) = C(n) + C(\sigma)$*, and* $C(\epsilon) = 0$

For example, the input sequences 0111 and 11100 both go to state s_5, but C(0111) = 8+2+2+2 = 14 and C(11100) = 2+2+2+8+8 = 22. The total cost of a learning process is the sum of the cost of all inputs made to the SUL plus the amount of resets needed times the reset cost.

3.3 Avenues to Reduce Cost

There are various ways to reduce the cost of learning a model. One idea is to change or augment the model learning algorithm to optimise for cost. One part of this could be to only reset if the cost of a reset is cheaper than returning to a previous state with normal inputs. However, optimisations in one algorithm are not likely to be easily transferable to other algorithms. While such improvements could be useful for a model learning algorithm, we will not be discussing them in this thesis.

A different approach to reducing learning cost would be to gather input-output-sequences before the main learning process starts. If these input-output-sequences are very long and avoid the expensive inputs, then it could be that the learner already has a lot of information to construct a decent model for relatively cheap.

In this thesis, we will be looking at optimisations within the testing phase of the learning process. Currently, most inputs are used within the testing phase as opposed to the learning phase[2, 20, 22]. This means that optimisations in the testing phase could have a greater effect than optimisations in the learning phase.

Additionally, optimisations in this space are very likely to be transferable between different model learning algorithms and will not require expert knowledge of that algorithm to implement.

The measures in this thesis can can be divided in two different approaches. The first approach is to optimise the individual segments that make up a test sequence. The second approach is to change the order in which we present potential counterexamples to the SUL.

Recall Section 2.4.3, which explained that a test sequence *piw* can be formed by taking a segment from the three sets P, I and W. The first approach is to optimise each of these segments to minimise cost. We start by constructing P with Dijkstra's shortest path algorithm. Next we try to find a cheaper segment from set I with a weighted random walk instead of a full random walk. Finally, we make the cheapest segment from W by applying Dijkstra's shortest path algorithm to a graph filled with Cartesian products of the set of states S.

For the second approach we first generate a list of test sequences, and then sort them in such a way that a 'good' counterexample will be tested earlier rather than later. Sorting methods implemented include by length, cost, cost per length, how often input symbols occur in the sequence, and cost per occurrence.

The next sections elaborate on each implemented method of reducing cost that will be tested.

3.3.1 Dijkstra P

Recall from Section 2.4.1 that set P is the set of access sequences for all states in the current hypothesis. Element $p_s \in P$ is then a specific input sequence to state s. To reach every state, only one sequence is needed per state in the hypothesis. In the L# learning library p_s is found with Breadth First Search (BFS). With BFS the access sequence length is minimised. However, cases exists where the shortest path is not the cheapest. The model in Figure 5 has two inputs, 1 and 0. The shortest path to state s_2 is by using input 0 once (pictured in red). Suppose that 0 is ten times more expensive than 1. Then we see that a cheaper, but also longer, path exists to state s_2, namely 1 twice.

In Figure 5, substituting the starting 0 with 11 in every access sequence could save even more costs as even the access sequences for states s_3 through s_6 would become cheaper. Thus, we would want to generate a set P that minimises cost instead of length.

To generate this P for our current hypothesis $(S, s_0, I, O, \delta, \lambda)$, we use Dijkstra's shortest path algorithm. For our purposes, we will say that Dijkstra works on a weighted graph $G : (V, E, d)$, where $v, v' \in V$ are the vertices, $e \in E$ are the edges (v, v') and $d : E \to N$ is the function that determines the distance between each node. We fill in this graph we say that $(V, E, d) = (S, E', w')$, where S is the set of states in our hypothesis and E' and w' are constructed as follows:

For every $s \in S$ and $i \in I$, we add an edge $e = (s, \delta(s, i))$ to set E', marked with input i. The cost for this edge is then calculated with $w'(e) = C(i)$.

Dijkstra's algorithm then returns a sequence of vertices visited, but we need the *edges* used to visit these vertices. Extracting this is not too hard. For every node pair s and s' that follow each other in the solution, we simply pick the cheapest edge. From which we can take the input-symbol.

Because the Dijkstra gives us the shortest path (based on cost) we also get the cheapest input sequence between the initial state and every other state.

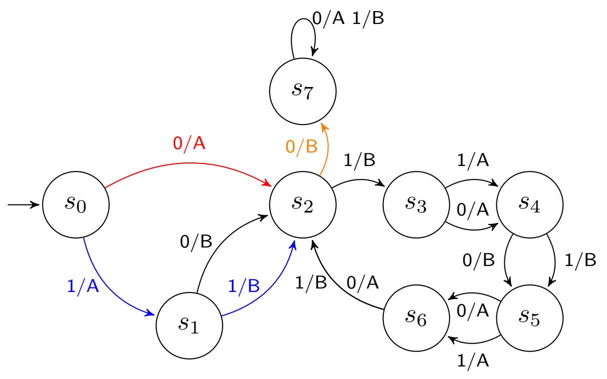


Figure 5: An example of a Mealy Machine

3.3.2 Weighted Random I

The i from I is a random walk of with a random length.

The value of the random length has an expected value that can be chosen through the command line input in the L# learning library. Specifics about the length of the random walk can be found in a paper by Garhewal and Damasceno[8]. In the L# learning library, every input has an equal chance of being picked every step of the random walk. The purpose of this sequence is to find a state or transition that is not yet in, or contradicts the latest hypothesis.

When one input has a higher cost we might want to minimise using that input in the random walk to save cost. However, a counterexample to the hypothesis might need that expensive input. A weighted random $i \in I$ could reduce costs for these cases.

Instead of choosing inputs from a uniform distribution, we instead choose from a weighted distribution, weighted on the inverse cost of the inputs. That is, cheaper inputs are chosen more often than expensive inputs. In my implementation, if an action is 10 times cheaper than another, then it is 10 times more likely to be chosen within the random walk. In our running example in Figure 5 the 0 input was ten times more expensive than the 1 input. Additionally, there are many transitions where you would rather do a 1 input than a 0 input. For example, you would not want to use a 0 input in s_2 (in orange) very often, because it leads to sink state s_7 where not much else would be learned. However, it could also make learning s_4 to s_6 harder, because it chooses input 1 more often, so it takes longer to find the correct output for input 0 in these states.

Weighted Random walk would likely be most useful when specific inputs can be avoided, like an input that always goes to an error state. The model in Figure 7 could benefit a lot from a weighted random walk. Inputs b_0 to b_n always go to a sink state and not any new states. If these inputs are expensive, they are less likely to be chosen and other more helpful inputs will be chosen instead.

In normal L#, if I has n different inputs, then each input has a $\frac{1}{n}$ chance of being chosen every step of the random walk. For the weighted random I we make it so every input i has a $\frac{\sum_{j \in I} C(j)}{C(i)}$ chance of being chosen. To give an example, when $C(0) = 10$ and $C(1) = 1$, we expect input 1 to be chosen ten times more often than input 0.

In cases where a more expensive input is crucial to the overall model it could happen that such an input is chosen so rarely that the overall cost of learning the model increases compared to a uniform random i. The learning algorithm only chooses cheap options that do not help it learn more about the target model. Take the following example in Figure 6. The only way to distinguish states s_b0 and s_b1 is by using input b1 or b0 twice in a row. Because the chance of high cost of these inputs twice is so low, that the learning process seems to stall.

Because of these two wildly different cases, it might be up to the domain expert to determine whether it is worth using this measure, or to switch back to normal i generation during the learning process, via some other method that detects that a weighted random walk is not effective.

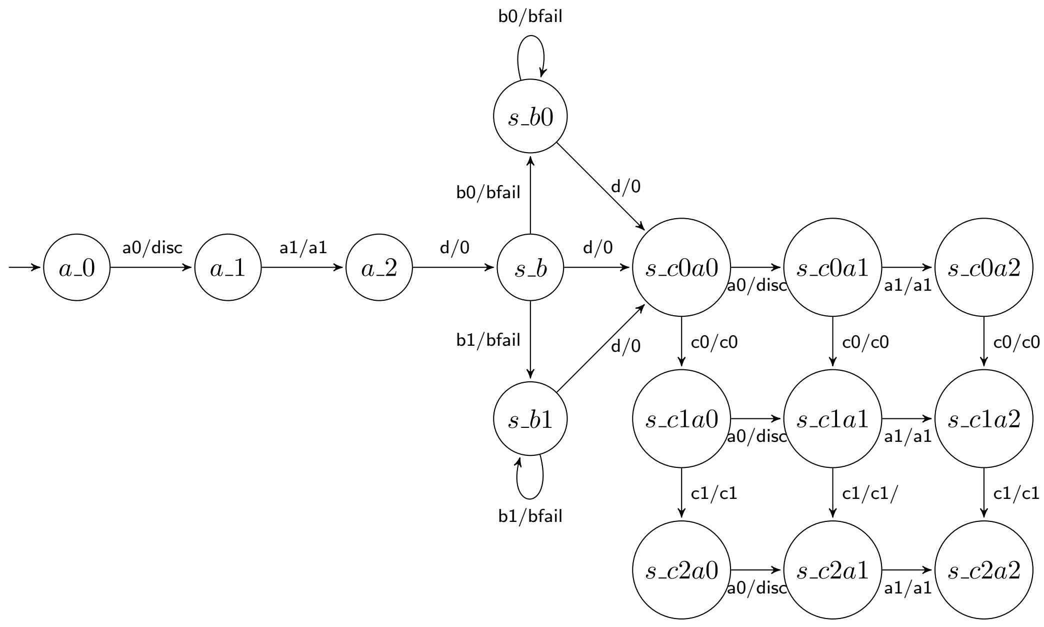


Figure 6: OpenSSH model generated with a script that makes models similar to models from the paper Parametrized Mealy Machines by Kruger et al.[10]. Inputs not pictured go to a sink state with an error output. This includes inputs $b0$ to b_n, that always go to this sink state.

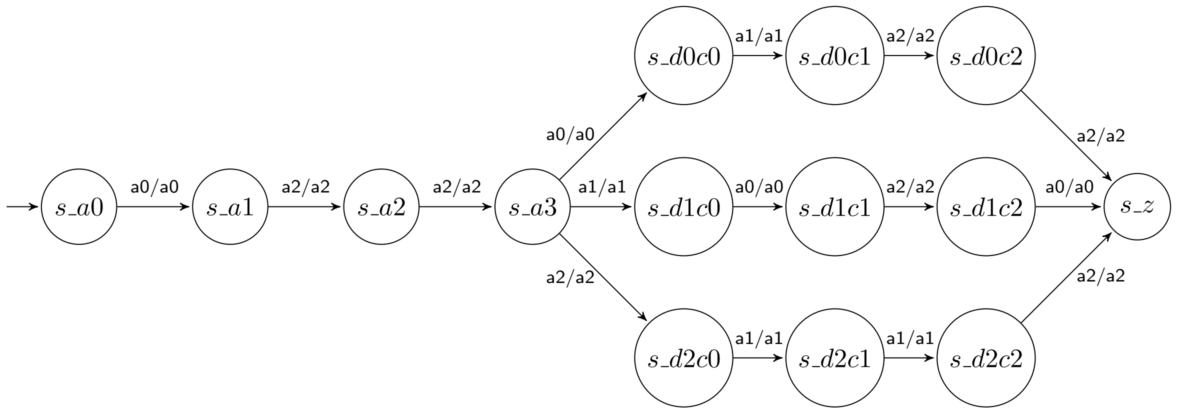


Figure 7: ASML model generated with a script that makes models similar to models from the paper Parametrized Mealy Machines by Kruger et al.[10]. Transitions not pictured go to a sink state with 'disc' as output.

3.3.3 Cartesian-Dijkstra W

The w from W is a separating sequence from the characterisation set W. This sequence *distinguishes* the state that the learner thinks we reached with the random walk from another state. Various options exist within the L# learning library for creating the characterisation set W. Instead of finding the shortest or most informative sequences that distinguishes every pair of states, we could find the sequences that have the lowest cost for each pair of states.

To find the cheapest separating sequence we can use Dijkstra again. This time the nodes in the graph will be each state-pair in the hypothesis: That is, the set of nodes V is equal to the Cartesian product $S \times S$, as well as one 'finish' node. To define the edges we have the following case distinction:

For every node $(s_1,\, s_2) \in V$, and all inputs $i \in I$ we add an edge $((s_1,\, s_2), s')$, where s' is decided as follows:

$$s' = \begin{cases} finish, & \text{for } \delta(s_1,i) \neq \delta(s_2,i) \\ (\lambda(s_1,i),\lambda(s_2,i)), & \text{for } \delta(s_1,i) = \delta(s_2,i) \wedge \lambda(s_1,i) \neq \lambda(s_2,i) \\ \uparrow & \text{for } \delta(s_1,i) = \delta(s_2,i) \wedge \lambda(s_1,i) = \lambda(s_2,i) \end{cases}$$

In cases where the outputs are different, then we have found a way to distinguish the two states s_1 and s_2 so we go to the 'finish' node. When the input has the same output for both s_1 and s_2, the edge goes to the node with the states that are reached when using input i in states s_1 and s_2. It is not possible to distinguish a state from itself, so when we the input reaches a node where $s_1 = s_2$, then we do not add the edge to the graph. The weight of each edge is simply the cost of the input. By starting from a node with the two states we want to distinguish, we can find the cheapest path to the finish node. When we have found the shortest path, then we extract the cheapest separating sequence.

To give an example, say we want to find a separating sequence for state-pair (s_1, s_2) in Figure 8. Just the input a (the transition in red) would do the trick, but a is ten times more expensive than b. With Cartesian-Dijkstra we can find the cheaper $\{b, b\}$ sequence (in blue). Using the process as described above, we can create the graph in Figure 9. In this figure we can see all nodes for each state-pair. Every node has an outgoing transition for each input-symbol, except when the output goes to (s_0, s_0), (s_1, s_1) or (s_2, s_2). Transitions with different outputs go to the finish state. For example, the red transition; an input a in s_1 gives an x output, but a y output in state s_2. This means we have found a way to distinguish these states, thus we add a transition to the finish state.

Transitions with equal outputs go to a different node in the graph. One of these transitions is the first b transition (in blue) from the cheap $\{b, b\}$ sequence we are looking for. The input b gives an

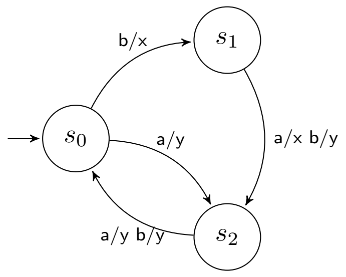


Figure 8: Cartesian-Dijkstra example model

output of y in both s_1 and s_2, meaning we cannot find a way to distinguish the two states directly. However, from the node we reach with this transition (s_2, s_0), we *can* reach the finish node, because b has different outputs from states s_2 and s_0. With Dijkstra we can find the $\{b, b\}$ sequence instead of the more expensive $\{a\}$ sequence.

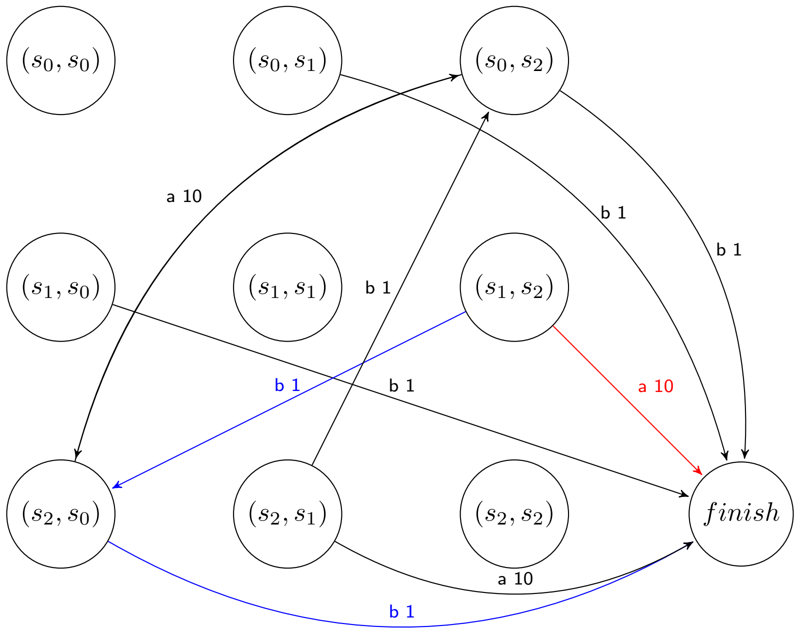


Figure 9: Cartesian-Dijkstra graph.

In practice, implementing this concept is a bit harder. The L# learning library uses Rust, which is a strongly typed language, meaning we cannot just add a 'finished' state to a set. Additionally, Dijkstra's output is a sequence of visited vertices, not the edges used to travel between the vertices.

To handle these problems I implemented the graph as a graph of $S \times S \times I$. In each vertex (s_n, s_m, i), s_n and s_m is the normal state-tuple and i is the input symbol used to get to this state-tuple. For every $i \in I$, (s_n, s_m, i) has the same outgoing edges, but only ingoing edges for input i with cost $C(i)$. If the output for input i is the same we ad an edge to vertex $(\lambda(s_n), \lambda(s_m), i)$. If $\lambda(s_n)$ and $\lambda(s_m)$ are equal, we do not add the edge, meaning that some vertices remain unused. If the output for input i is different, we cannot go to a 'finish' state with a different type, so instead we go to the unused vertex $(s0, s0, i)$. Vertices where $s1$ and $s2$ are the same where unused because it is impossible to find a separating sequence. Finally, we have one more transition to state $(s0, s0, i0)$ with distance 0 to finalize the route. We can then extract the input sequence by taking the i from every vertex visited. In the case that $i \neq i0$, we remove the last input symbol.

Now we have the cheapest separating sequence for every state-pair. However, this still might not save costs compared to existing techniques. Some methods like ADS try to find a single separating sequence for each state. Cartesian-Dijkstra makes multiple sequences per state, unless it coincidentally made the same sequence for different states. The test sequence generator then has to choose one. In cases where only a single sequence could find the counterexample, having more separating sequences to choose from reduces the chance of picking the sequence we need. This means we need more test rounds and thus incur more costs. Sadly I did not have time for this thesis to find a method that finds a balance between the amount of separating sequences per state and a minimal cost per state-pair.

3.3.4 Pre-Generating Sequences

In this section I explain how pre-generating sequences and then ordering them in some way could help with reducing costs.

As the running example for this section we use the hypothesis made using the L# learning library while learning a model based on the model in Figure 10. This model is based on the 'OpenSSL_1.0.1l_client_regular.dot' model from the test model set used in the original L# paper[24][3]. To get to the last state s_4, you need a very specific input sequence, ending with C. This C input is ten times more expensive than the other inputs, so we would like to avoid the C input as long as possible.

To find a counterexample, we generate ten different sequences, all from PIW. There are various ways to sort this list. These can be seen in Table 1. We will go over the various ways of sorting the PIW list in this section with the sequences in this table as an example.

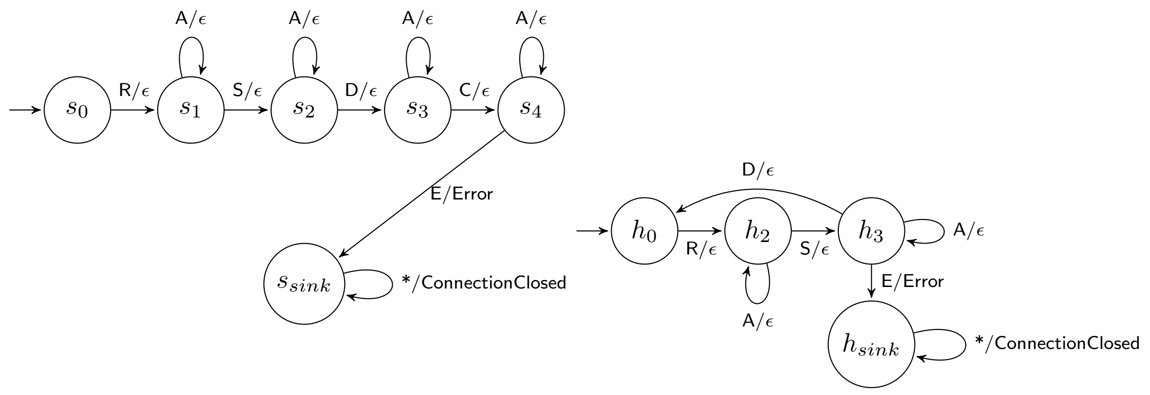


Figure 10: Example target Mealy Machine (left) and the hypothesis created by L# (right). Transitions not pictured give an Error output and go to the sink in both the target and the hypothesis.

Sequence	Length	Sequence	Cost	Sequence	Cost per Input	Sequence	Occurrence	Sequence	Cost per Occurrence
S, S, D, C, E	5	R, D, E, A, E, E	6	R, D, E, A, E, E	1	A, A, C, F, C, E	8	A, A, C, F, C, E	0.33
A, D, S, C, C, E	6	R, S, D, E, R, E	6	R, S, D, E, R, E	1	R, R, F, A, C, E	9	R, S, D, E, R, E	0.5
A, S, S, R, C, E	6	R, S, S, S, A, F, E	7	R, S, S, S, A, F, E	1	S, S, D, C, E	9	R, D, E, A, E, E	0.5
R, D, E, A, E, E	6	S, S, D, C, E	14	R, S, D, C, D, A, E	2.29	A, S, S, R, C, E	11	R, S, S, S, A, F, E	0.58
R, S, D, E, R, E	6	A, S, S, R, C, E	15	R, S, R, R, C, F, E	2.29	R, S, R, R, C, F, E	11	R, R, F, A, C, E	0.6
A, A, C, F, C, E	6	R, R, F, A, C, E	15	A, S, S, R, C, E	2.5	A, D, S, C, C, E	10	R, S, D, C, D, A, E	1.23
R, R, F, A, C, E	6	R, S, D, C, D, A, E	16	R, R, F, A, C, E	2.5	R, S, S, S, A, F, E	12	A, S, S, R, C, E	1.36
R, S, D, C, D, A, E	7	R, S, R, R, C, F, E	16	S, S, D, C, E	2.8	R, S, D, E, R, E	12	R, S, R, R, C, F, E	1.45
R, S, S, S, A, F, E	7	A, D, S, C, C, E	24	A, D, S, C, C, E	4	R, D, E, A, E, E	12	S, S, D, C, E	1.6
R, S, R, R, C, F, E	7	A, A, C, F, C, E	24	A, A, C, F, C, E	4	R, S, D, C, D, A, E	13	A, D, S, C, C, E	2.4

Table 1: Various ways of sorting the 10 generated test sequences.

The easiest way of sorting the list is by the **length** of the sequence. For this example we will be sorting from shortest to longest, although both ways are possible. As can be seen in the first column of Table 1, the shortest sequence is S,S,D,C,E. We continue down the list until we try R,S,D,E,R,E where the hypothesis has a different output sequence than the SUL. In total, it has cost us 63 to find this counterexample. But if we had started from the other side of the list, we would have found R,S,D,C,D,A,E for a smaller cost of 39. Long test sequences might have more information. Despite having a higher chance to be more expensive, they could cause the learning process as a whole to be cheaper overall.

We could also sort by **cost**. By doing the cheaper sequences first, we hope to save total costs by finding a cheap counterexample first, or exhaust a bunch of cheap test sequences before considering sequences that are more expensive. For every test sequence $\sigma \in I$ we calculate the cost $\mathrm{C}(\sigma)$ and order them from low to high. In the example from Table 1, it would only cost 12 to find counterexample R,S,D,E,R,E, as the cheap counterexample. The shorter but more expensive S,S,D,C,E is skipped compared to the ordering by length.

Sorting by cost biases the sequences toward shorter sequences when most inputs cost the same low amount. In cases where longer counterexamples could help more, it might be better to compensate for this bias by dividing the cost by the length of the sequence. We are then sorting by **cost per length**. For every test sequence $\sigma \in I$ we calculate the value $\frac{C(\sigma)}{|\sigma|}$. Longer sequences could contain more information to learn from, but are more expensive. By sorting with this, we hope to find a balance between the length and cost. In this specific example it does not change anything compared to sorting by cost, but you can see how S,S,D,C,E, the shortest sequence, is almost at the bottom of the list because it contains the expensive input C.

We can also sort by what I will call **occurrence**. Instead of cost, we add up how often each input symbol has already appeared in previous counterexamples. For example, in models like in Figure 10, you are expected to take a certain path through the SUL with specific inputs at specific states. It could be beneficial to promote the use of input symbols that have appeared in previous counterexamples to increase the chance of reaching a state that is farther along in the model. For every input symbol $i \in I$, we track how often it appears in the found counterexamples. Let $Occ : I \to \mathbb{N}$ be the occurrence function. $Occ(i)$ denotes how often symbol i appears in the counterexamples. The value for test sequence $\sigma = i^0 \dots i^n$ is $\sum_{i \in \sigma} Occ(i)$. When we sort from high to low, sequences that contain symbols that have appeared in a lot of counterexamples will be tested first. In this case, C has only been in a counterexample once before, while F was never in a counterexample. The other symbols where in two counterexamples. If we go from the bottom, we find the two possible counterexamples very quickly, because they contain many symbols that have been used before, and that are needed to get deeper in the model. The opposite way of sorting could be useful for models that split early, so you can explore new nodes in unexplored parts sooner.

You can also sort by **cost per occurrence**. You can compensate for the cost of a sequence with how often the symbol appears in counterexamples. Maybe a symbol is helpful in finding counterexamples, but also expensive. This ordering attempts to balance the cost with occurrence. It is calculated with $\frac{C(\sigma)}{Occ(\sigma)}$. If a costly symbol is discouraged by cost, but is often used by counterexamples anyway, then this way of sorting might find a better balance between the two. In this case it did not change much, as this hypothesis was from fairly early in the learning process.

For this example, both **cost per length** and **cost** would be best to find a counterexample for the least amount cost. We will see if these are still good ways of reducing cost in the full learning process in the next section where we compare all measures with normal L#.

4 Experiments

We want answers to the following questions to see whether the implemented measures have had an effect:

R1 How effective is a test sequence with a cost-minimal P at reducing learning cost?

R2 How effective is a test sequence with a weighted I at reducing learning cost?

R3 How effective is a test sequence with a state-pairwise cost-minimal W at reducing learning cost?

R4 How effective is a test sequence with a cost-minimal P, a weighted I and a state-pairwise cost-minimal W at reducing learning cost?

R5 How effective is sorting a pre-generated list of test sequence at reducing learning cost?

For 'effectiveness' we will be looking at how cost was saved compared to normal L#, like whether the measure saves cost in general over multiple models, whether the cost increases greatly for specific models, or whether costs decrease greatly for specific models.

Each question will be answered in three different contexts. The three different contexts are:

- Full run with random weights.
- Finding a counterexample to a hypothesis.
- Full run with predefined weights.

To answer these questions we perform the experiments as described in the next section.

4.1 Experiment Setup

To see whether a measure saves costs overall, we compare full runs with and without the measures on multiple models with random weightfiles. A full run meaning we run the L# algorithm to learn a model once. We consider 'normal' or 'base' L# to be L# where P is generated with BFS, I with uniform random distribution, W with Hybrid-ADS. Results of experiments with these parameters will be considered the 'baseline'.

The models we will be learning were also used in the original L# paper [24][11] and are from the Automata Wiki[3]. Additionally, some models were generated with a script based on models described in Kruger et al.'s 'Small Test Suites for Active Automata Learning'[10].

To see whether a measure saves costs finding a counterexample, we use the L# algorithm with and without the measure to find a single counterexample for a hypothesis for each model.

We also compare full runs with and without measures, but with pre-defined weights instead of random weights. These weights will be chosen to simulate a domain expert helping the algorithm by providing higher weights to expensive inputs, or inputs that are not used much.

This means we do three experiments per measure. Details that pertain to all experiments will be elaborated upon in the sections below. Details for each individual experiment will be given in the section for that experiment.

4.1.1 Generating Random Weightfiles

The cost for each input(and the reset) is specified in the 'weightfile'. This choice has a large effect on the total cost, even more so than the initial seed. As such, 50 randomly generated weightfiles will be generated for most models. Larger models will have 30 weightfiles to save time and resources. Table 2 contains information on which models used how many seeds and weightfiles.

When generating a weightfile a few inputs are chosen as the 'expensive ones'. Inputs with undefined weights will be given a cost of one, expensive inputs are defined in the weightfile with a cost of 10. We automatically generate the weightfiles as needed during each experiment. These weightfiles are generated as follows:

```
1. We pick between 2 to $(|I|/2+1)$ inputs.
2. If there are only 2 or 3 inputs, then 1-2 inputs are chosen.
3. These inputs are put in a new weigthfile with the ‘high cost’.
4. The high cost defined for the experiment is 10.
5. The other inputs are not put in the weightfile, and are thus given the
   default cost of ‘1’. This includes the RESET cost.
```

Instead of giving all inputs a random value within a certain range, we only give a few inputs a single ‘expensive’ value. We do this because the goal of this thesis is to reduce cost in cases where a few inputs are more expensive than others. This way of generating a weightfile more closely resembles that situation than the alternative of giving each input a random weight within a pre-determined range.

4.1.2 Seeds

The random choices made during the learning process can greatly influence the total cost needed. To mitigate the effects of random chance we repeat the experiments many times on different seeds. However, we already use multiple weightfiles and using multiple seeds per weightfile means we have a greatly increasing amount of experiments to do. Because computing time is limited, we use fewer seeds for larger models (more than 30 states). Table 2 contains information on which models used how many seeds and weightfiles.

4.1.3 Hypothesis Extraction

To test the finding of the counterexample for a single hypothesis, I took the third hypothesis generated by a normal L# run. However, for small models the third hypothesis could be the complete target model. In those cases I took the second hypothesis. This only happened with models ‘4_learnresult_SecureCode_Aut_fix’ and ‘ASN_learnresult_SecureCode_Aut_fix’. Most hypotheses were 3 to 5 states large. Larger models had larger hypotheses. The largest hypothesis was from ‘model3.dot’, with a size of 29. ‘model1.dot’ made second place with a size of 12 states.

4.1.4 Summary

For each variable in the measure, we use the L# algorithm to learn each model. We do this (50 or 30) times for each seed, and (50 or 30) times for each weightfile, for a total of (2500 or 900) runs per model. Additionally, we do the same for finding a counterexample for a hypothesis for each model. Finally, we predefine a weightfile for each model, and run the algorithm (50 or 30) times for each seed with that pre-defined weightfile for each model.

4.2 Experiment 1: Dijkstra vs BFS

In this section we will be answering the question: 'How effective is a test sequence with a cost-minimal p at reducing learning cost?'

To answer the question we learn models as described above. First we do a full run of base L#, where access sequences in set P are generated with BFS. Then we learn the same models, but set P is generated with Dijkstra instead. Weights will be generated randomly as described above.

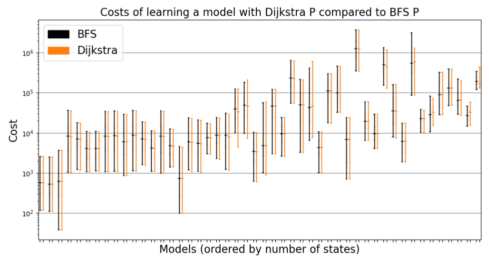
For the second part we do not learn an entire model, but try to find a counterexample to a hypothesis. Weights will be generated randomly once again.

Lastly, we use pre-defined weights instead of random weights to learn a smaller selection of models.

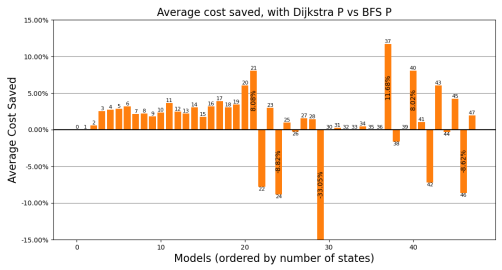
4.2.1 Randomly Generated Weights

Figure 11a shows box-plots of the cost incurred during the testing phase for each model where either BFS or Dijkstra was used to generate the P set. The models are sorted by the amount of states in each model. The top whiskers show the 5 and 95th percentile cost, the middle denotes the median cost.

Figure 11b shows how much cost was saved on average in the testing phase. This was calculated by, for each model, taking the average of all BFS costs and subtracting the average of all Dijkstra cost.



(a) Range and mediums of costs for learning models with BFS and Dijkstra. Whiskers are on the 5th and 95th percentile cost for each model.



(b) Average cost saved using Dijkstra compared to BFS as a percentage. Calculated by averaging all costs for Dijkstra and all costs for BFS for each model, then interpreting BFS as 100%.

We can see in general that cost did not increase significantly and oftentimes, cost was saved for most models learned. Take model 5 for instance. Model 5 is the 'OpenSSL_1.0.2_client_regular.dot' model. This model is similar to the one pictured in Figure 10, where many transitions go to a sink state, while the rest of the model is a straight line with some self-loops. Even without any alternative paths to non-sink states some cost was saved, mostly by avoiding the expensive transitions.

There also a few models where a lot of extra cost was incurred. In model 29 ('NSS_3.17.4_client_full', pictured in Figure 12) this went up to 31%. The reason for the extra cost can be found within the randomly generated weightfiles. The average cost when the 'ServerHelloDHE' input was chosen as the expensive input is much higher than when it was not chosen, but only when input 'ServerHelloRSA' was *not* chosen to be expensive. The target model in Figure 12 can show us why this might be the case.

In early hypotheses, weighted L# considers states s_1a and s_1b equivalent. Once weighted L# makes this assumption it will only use 'HelloRSA' to move to state s_1a/s_1b and beyond, because it thinks it is the strictly cheaper option. Test sequences that correct this assumption are very rare. First an access sequence needs to be chosen to the initial state, then, in the random walk and distinguishing sequence, L# would need to choose 'HelloDHE', 'Certificate', 'CertReq' and 'ChangeCipherSpec' in that order. The chance of this test sequence generating is very low, and thus it will likely take a long time (and cost) before the false assumption is fixed. If 'HelloRSA' and 'HelloDHE' have the same cost, then either can be chosen in the access sequence, which avoids the pitfall.

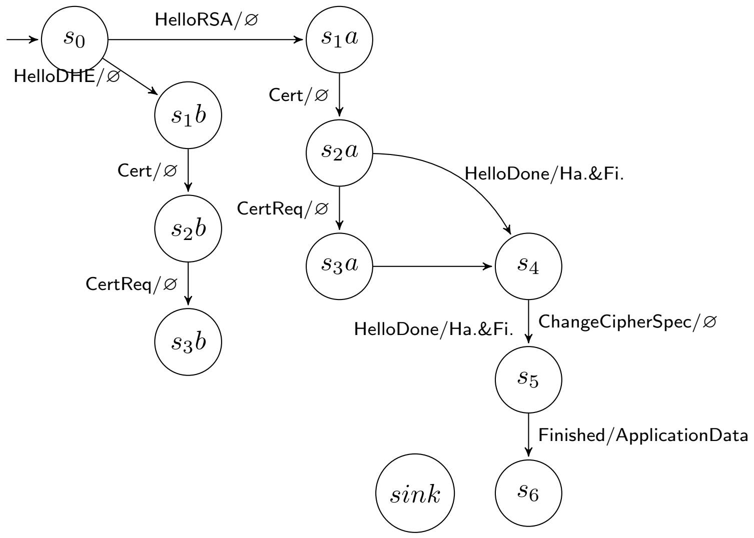
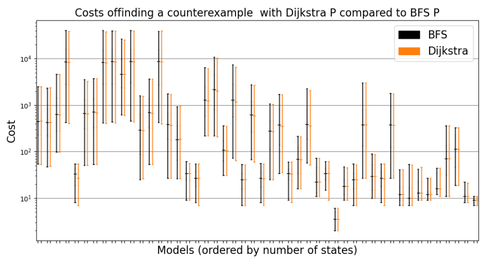


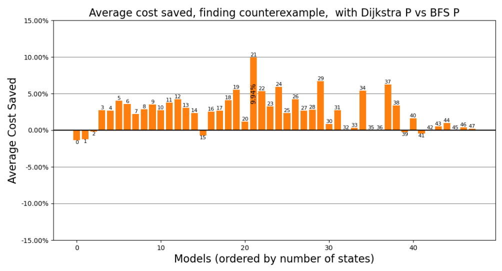
Figure 12: Simplified NSS model. Input-names have been abbreviated and self-loops and transitions with error outputs are not pictured

4.2.2 Finding a Counterexample

Figure 13a shows the ranges of possible costs for each model. In general, the ranges of BFS and Dijkstra are fairly similar. Figure 13b shows that overall, using Dijkstra does save some costs on average. There are very few models where Dijkstra was not on average cheaper than BFS, including model 29 that Dijkstra had problems with in the full learning process. Because the hypothesis was from earlier in the learning process, the issue of model 29 did not appear yet.



(a) Range and mediums of costs for finding a counterexample with BFS and Dijkstra. Whiskers are on the 5th and 95th percentile cost for each model.

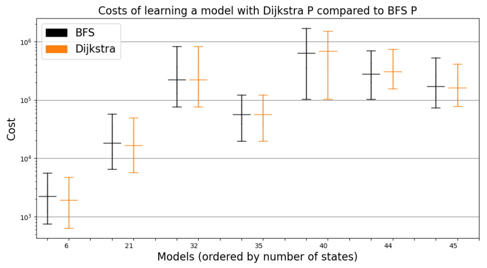


(b) Average cost saved using Dijkstra compared to BFS as a percentage when finding a counterexample. Calculated by averaging all costs for Dijkstra and all costs for BFS for each model, then interpreting BFS as 100%.

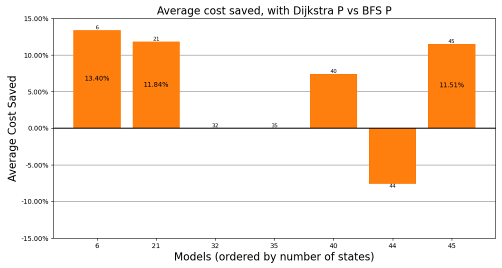
4.2.3 Predefined Weights

Figure 14a and 14b show results for full runs with pre-defined weights.

In this case, there not too many drastic changes compared to the results of the random weight full runs. Learning model 6 was a bit cheaper, but learning model 21 with the same input-costs made it a bit more expensive on average than with the random weights. These results mostly just echo the results of the random weights experiment. When the expensive input is used in test sequences in models 32 and 35, there is no alternative input that could make the sequence cheaper, so no cost could be saved.



(a) Range and mediums of costs for learning models with BFS and Dijkstra. Whiskers are on the 5th and 95th percentile cost for each model.



(b) Average cost saved using Dijkstra compared to BFS as a percentage. Calculated by averaging all costs for Dijkstra and all costs for BFS for each model, then interpreting BFS as 100%.

4.3 Experiment 2: Random Walk vs Weighted Random Walk

In this section we will be answering the question: 'How effective is a test sequence with a weighted i at reducing learning cost?'

To answer the question we learn models as described above. We use L# to learn the models, but the random walk i is generated with either a uniform distribution (Random) like normal L# or a weighted distribution (Weighted). First we do a full run comparing Random and Weighted, with randomly generated weights.

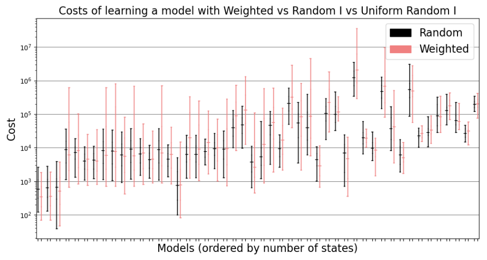
Second, we do not learn an entire model, but try to find a counterexample to a hypothesis. Weights will be generated randomly once again.

Lastly, we use pre-defined weights in instead of random weights to learn a smaller selection of models.

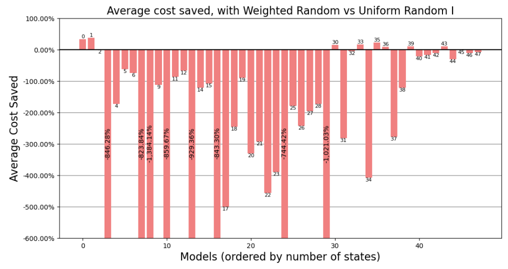
4.3.1 Randomly Generated Weights

Figure 15a shows the ranges of costs learning different models between Random Walk and Weighted Random Walk as box-plots, with horizontal markings showing the 5th and 95th percentile as well as the median.

Weighted Random has a much larger range of possible cost compared to Uniform Random. Sometimes the median is lower with Weighted than with Random, for example the first four models. However, often the top whisker goes much higher than the bottom whisker goes lower, so the average cost could result to be higher on average.



(a) Range and mediums of costs for learning models with uniform random or weighted random walk. Whiskers are on the 5th and 95th percentile cost for each model.

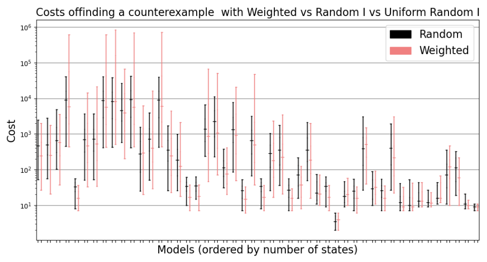


(b) Average cost saved using weighted random compared to a uniform random as a percentage. Calculated by averaging all costs for each method for each model, then interpreting uniform random results as 100%.

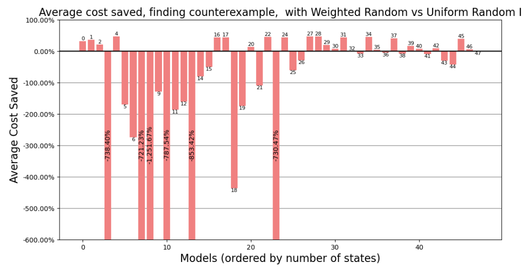
This is indeed what we see in Figure 15b. Using Weighted random seems to be a bad idea as costs are more likely to increase dramatically. Often at twice as expensive (at the -100% line), with some models reaching 10 times the normal cost. The model reaching 1391.75% is '4_learnresult_PIN_fix.dot'.

4.3.2 Counterexample Searching

Despite the higher cost seen in full model learning experiment, finding a counterexample to a hypothesis seems to be less overall bad for the weighted approach when seen from the average cost saved graph (16b). While there are still models where costs balloon, some models save almost 50% of cost. When seen from the box-plot (Figure 16a), we see that costs for finding a counterexample in these models were low to begin with (below 100). This is likely because finding a counterexample was easy to begin with. Many sequences would qualify, the change with Weighted, is that cheaper sequences are chosen.



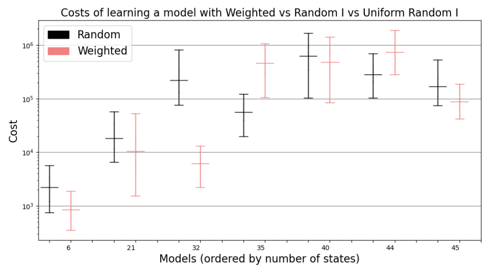
(a) Range and mediums of costs for finding a counterexample with uniform random or weighted random walk. Whiskers are on the 5th and 95th percentile cost for each model.



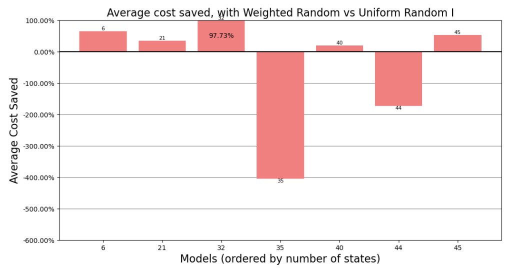
(b) Average cost saved finding a counterexample using weighted random compared to a uniform random as a percentage. Calculated by averaging all costs for each method for each model, then interpreting uniform random results as 100%.

4.3.3 Predetermined Weights

Figure 17a and 17b show results for full runs with weights chosen by me. Interestingly, these results show large differences with respect to the random weightfile runs. Using weighted random with random input weights had more costs compared to uniform random for 5 of the 7 models. But with pre-defined weights, all of those saved costs instead, massively so for model 32 ('ASML_3_3_3_3.dot'). This fits the prediction made in Section 3.3.2. This model has many inputs that are not used for finding new states. Avoiding these massively reduced cost. On the other hand using the new measures made learning model 35 and 44 more expensive than before. Unlike model 32, model 35 ('OpenSSH_2_2_2.dot') does require use of the expensive inputs to reach and distinguish new states. As described in Section 3.3.2, this model is more expensive to learn because the chance of two expensive inputs being chosen in a certain state within a test sequence is very low.



(a) Range and mediums of costs for finding a counterexample with uniform random or weighted random walk. Whiskers are on the 5th and 95th percentile cost for each model.



(b) Average cost saved finding a counterexample using weighted random compared to a uniform random as a percentage. Calculated by averaging all costs for each method for each model, then interpreting uniform random results as 100%.

4.4 Experiment 3: Hybrid-ADS vs Wp vs Cartesian-Dijkstra

In this section we will be answering the question: 'How effective is a test sequence with a state-pairwise cost-minimal w at reducing learning cost?'

To answer the question we learn models as described above. This time we learn models where the characterisation set is generated with the Wp-method, Hybrid-ADS method and the new Cartesian-Dijkstra method.

First we do full runs comparing the three methods of generating the characterisation set W, with randomly generated weights.

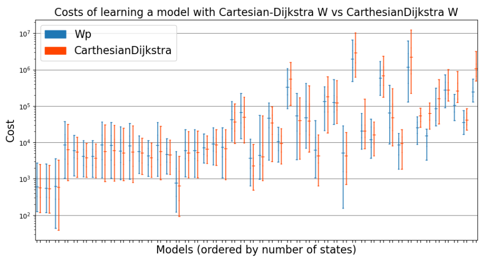
Second, we do not learn an entire model, but try to find a counterexample to a hypothesis. Weights will be generated randomly once again.

Lastly, we use pre-defined weights instead of random weights to learn a smaller selection of models.

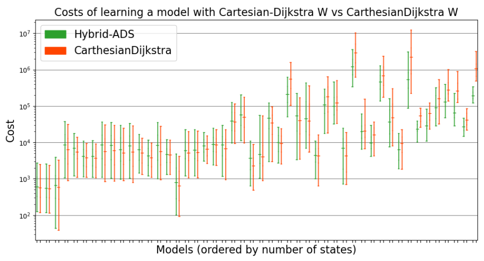
4.4.1 Randomly Generated Weights

In Subfigure 18c and 18d we see that for most smaller models Cartesian-Dijkstra on average saved cost compared to both Wp and Hybrid-ADS. However, for both methods, using Wp and Hybrid-ADS is still cheaper for larger models. Quite dramatically so in the case of Hybrid-ADS. Cartesian-Dijkstra did best compared to Wp when learning model 30 (TCP_FreeBSD_Client.dot), but Hybrid-ADS managed to save even more cost than Cartesian-Dijkstra. Hybrid-ADS makes smaller W sets, increasing the chance of the 'correct' separating sequence being chosen. This aspect matters most when learning a model takes more rounds or when counterexamples are rarer. This is likely why you would still prefer to use Wp or Hybrid-ADS when learning the larger models.

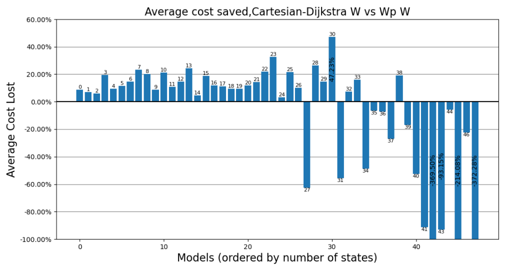
Cartesian-Dijkstra vs Wp and HADS-INT, Random Weightfiles



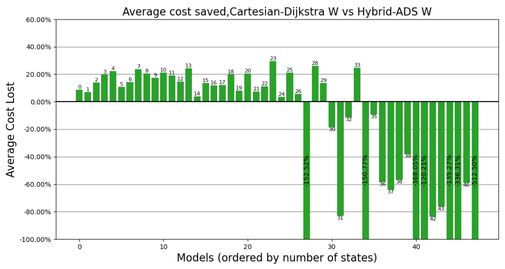
(a) Range and mediums of costs for learning models with the Wp or Cartesian-Dijkstra method. Whiskers are on the 5th and 95th percentile cost for each model.



(b) Range and mediums of costs for learning models with the Hybrid-ADS or Cartesian-Dijkstra method. Whiskers are on the 5th and 95th percentile cost for each model.



(c) Average cost saved using Cartesian-Dijkstra compared to the Wp method as a percentage. Calculated by averaging all costs for Cartesian-Dijkstra and all costs for Wp for each model, then interpreting Wp as 100%.



(d) Average cost saved using Cartesian-Dijkstra compared to the Hybrid-ADS method as a percentage. Calculated by averaging all costs for Cartesian-Dijkstra and all costs for Hybrid-ADS for each model, then interpreting Hybrid-ADS as 100%

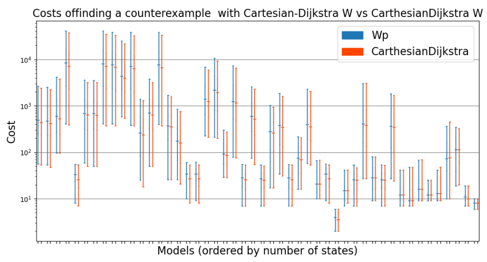
4.4.2 Counterexample Searching

When searching for a single counterexample, Cartesian-Dijkstra does outperform both Wp and Hybrid-ADS on most models. This is likely because in the earlier phases of the learning process, it does not take as many tries to find a counterexample as in later phases, where sometimes a very specific counterexample must be found. Before this point is reached, reducing the cost of the separating sequence does help with reducing the cost of finding a counterexample.

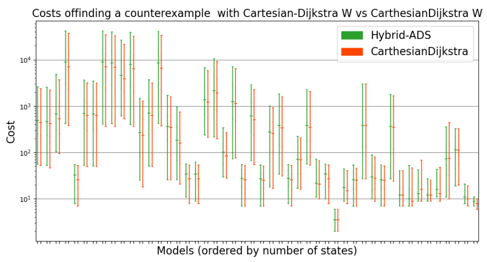
4.4.3 Predetermined Weights

Results mostly echo the results of the random weight tests. One thing of note is that the cost difference of learning the TCP server models was larger with pre-determined weights than with random weights, either because the weights were not fitting for the model or that this measure was not helpful to a realistic circumstance.

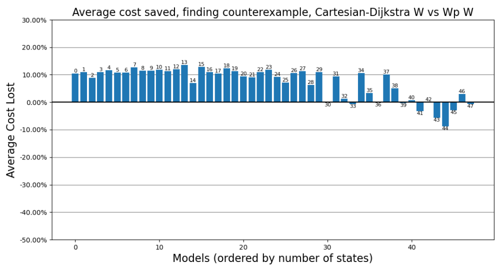
Cartesian-Dijkstra vs Wp and Hybrid-ADS, Finding counterexample to hypothesis



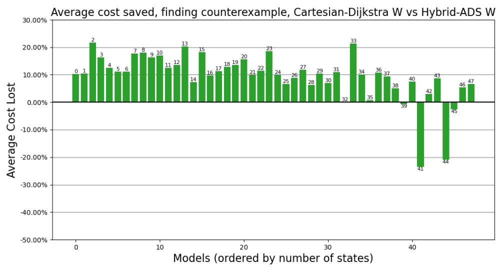
(a) Range and mediums of costs for finding a counterexample with Cartesian-Dijkstra or Wp. Whiskers are on the 5th and 95th percentile cost for each model.



(b) Range and mediums of costs for finding a counterexample with Cartesian-Dijkstra or Hybrid-ADS. Whiskers are on the 5th and 95th percentile cost for each model.

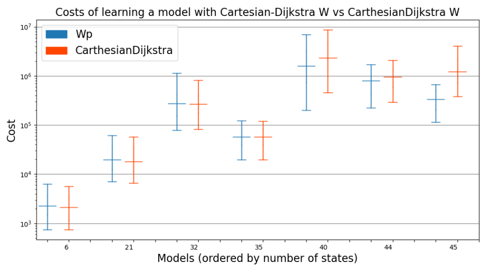


(c) Average cost saved finding a counterexample using Cartesian-Dijkstra or Wp as a percentage. Calculated by averaging all costs for each method for each model, then interpreting the Wp results as 100%.

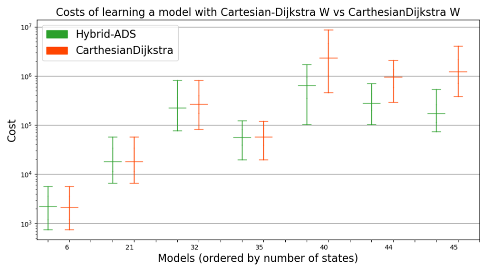


(d) Average cost saved finding a counterexample using Cartesian-Dijkstra or Hybrid-ADS as a percentage. Calculated by averaging all costs for each method for each model, then interpreting the Hybrid-ADS results as 100%.

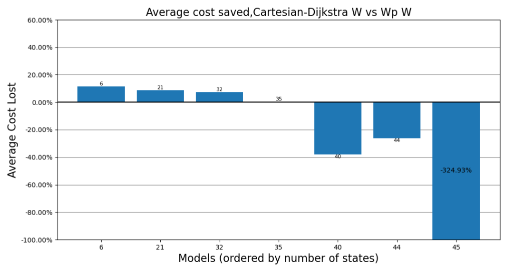
Cartesian-Dijkstra vs Wp and Hybrid-ADS, Pre-defined Weightfiles



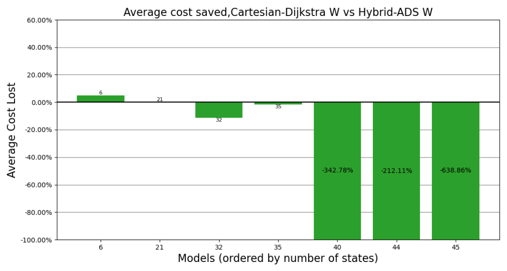
(a) Range and mediums of costs for learning models with the Wp or Cartesian-Dijkstra method. Whiskers are on the 5th and 95th percentile cost for each model.



(b) Range and mediums of costs for learning models with the Hybrid-ADS or Cartesian-Dijkstra method. Whiskers are on the 5th and 95th percentile cost for each model.



(c) Average cost saved using Cartesian-Dijkstra compared to the Wp method as a percentage. Calculated by averaging all costs for Cartesian-Dijkstra and all costs for Wp for each model, then interpreting Wp as 100%.



(d) Average cost saved using Cartesian-Dijkstra compared to the Hybrid-ADS method as a percentage. Calculated by averaging all costs for Cartesian-Dijkstra and all costs for Hybrid-ADS for each model, then interpreting Hybrid-ADS as 100%

4.5 Experiment 4: CHEAP PIW vs NORMAL

In this section we will be answering the question: 'How effective is a test sequence with a cost-minimal P , a weighted I and a state-pairwise cost-minimal W at reducing learning cost?'

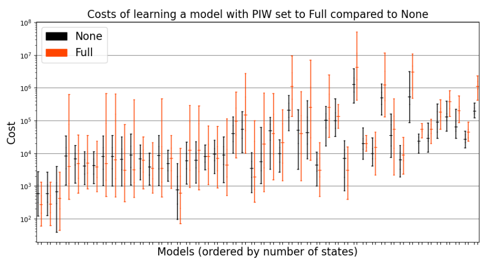
To answer the question we learn models as described above. This time we learn models with all three previous measures active and compare it with normal L#. When the P, I and W are generated with BFS, a uniform random distribution and Hybrid-ADS we call it 'Normal PIW'. When they are generated with Dijkstra, a weighted random distribution and Cartesian-Dijkstra we call it 'Cheap PIW'

First we do a full runs with randomly generated weights, comparing Normal PIW and Cheap PIW. Second, we do not learn an entire model, but try to find a counterexample to a hypothesis. Weights will be generated randomly once again. Lastly, we use pre-defined weights in instead of random weights to learn a smaller selection of models.

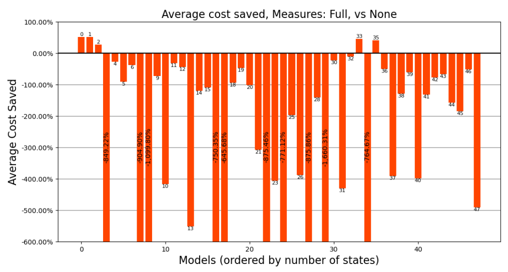
4.5.1 Randomly Generated Weights

Figure 21a shows a box-plot for all costs per model for Normal L# against Weighted L# with the cheaper PIW options enabled. Figure 21b shows the average cost saved. We see that most models do not benefit from all PIW measures being turned on at once. It seems to combine the worst aspects of each measure, like the spike on model 8. This is the same spike as in Figure 15b for the Random vs Weighted experiment. The spike on model 29 can also be seen in the Dijkstra vs BFS experiment in Figure 11b as well as in Figure 15b. They now combine to form an even larger cost sink.

Cheap PIW vs Normal PIW, Random weights



(a) Range and mediums of costs for learning models with and without measures. Whiskers are on the 5th and 95th percentile cost for each model.



(b) Average cost saved with and without measures as a percentage. Calculated by averaging all costs for each method for each model, then interpreting the results without measures as 100%.

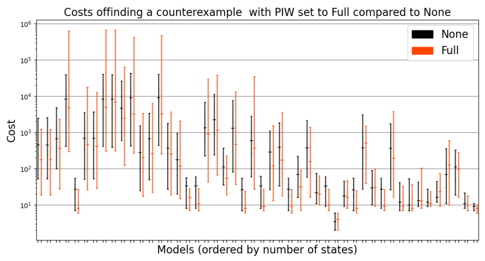
4.5.2 Counterexample Searching

The results for finding a counterexample to a hypothesis show mixed results. Figure 22a shows that for smaller models, where the hypothesis is already close to the full model, costs are likely to increase. Some models do show a decrease in cost, but these are generally models where costs to find a counterexample were low to begin with, as can be seen in Figure 22a. Some exceptions to this are models 0, 1, 3 or 20.

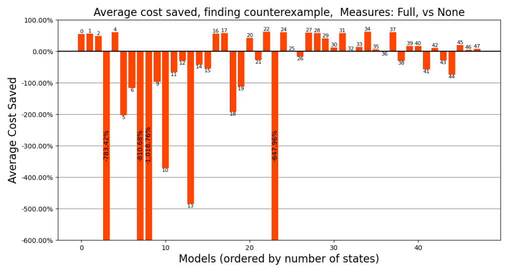
4.5.3 Predetermined Weights

We see a big difference with pre-determined weights compared to random weights. With random weights, the smaller models 6, 21 and 32 were more expensive to learn with the measures than base L#, but with pre-determined weights this turned around completely, now saving costs instead. On the other hand, learning model 35 became more expensive rather than less expensive.

Cheap PIW vs Normal PIW, Counterexample searching

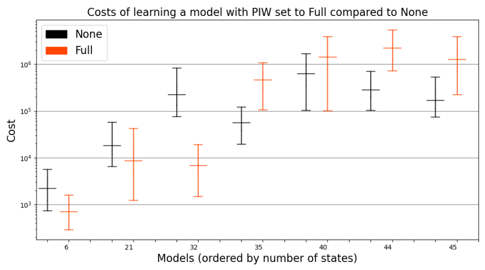


(a) Range and mediums of costs for finding a counterexample with and without measures. Whiskers are on the 5th and 95th percentile cost for each model.

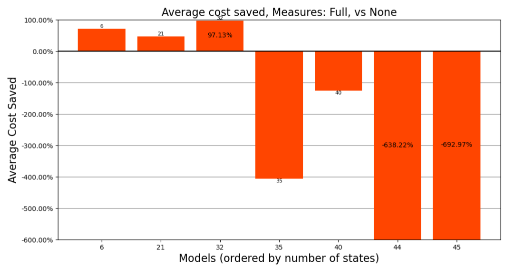


(b) Average cost saved finding a counterexample with and without measures as a percentage. Calculated by averaging all costs for each method for each model, then interpreting the results without measures as 100%.

Cheap PIW vs Normal PIW, Pre-defined weights



(a) Range and mediums of costs for learning models with and without measures. Whiskers are on the 5th and 95th percentile cost for each model.



(b) Average cost saved with and without measures as a percentage. Calculated by averaging all costs for each method for each model, then interpreting the results without measures as 100%.

4.6 Experiment 5: SORT vs NORMAL

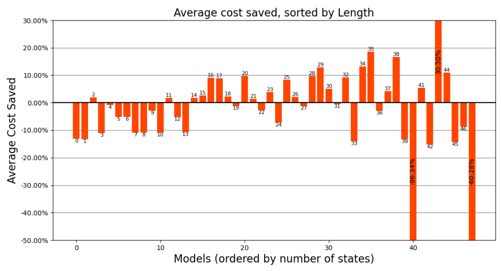
In this section we will be answering the question: 'How effective is sorting a pre-generated list of test sequence at reducing learning cost?'

To answer this question we learn all models with normal L#, where one test sequence is generated at a time, and by generating 20 sequences at once and then sorting them by cost, by occurrence, by cost-reversed, by occurrence-reversed, by cost-per-occurrence, by occurrence-per-cost, by length, by length-reversed, by cost-per-length or by length-per-cost. Once these 20 are tried and no counterexamples are found, a new set of 20 test sequences is generated.

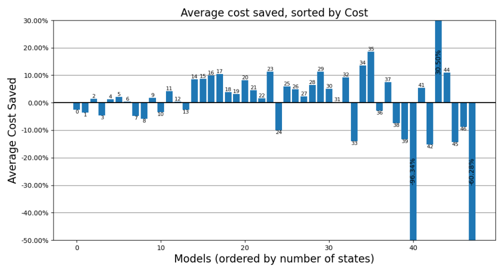
We use a list size of 20 instead of more or all because generating many test sequences risks that all counterexamples congregate at the bottom of the list, massively increasing the time it takes to find a counterexample. A smaller list size increases test sequence diversity, but lowers the effect size of the sorting method. 20 was guessed as a midpoint between test sequence diversity and sorting method effect size. Further testing should be done to find what the best list size is.

First we do a full runs with randomly generated weights. Second, we do not learn an entire model, but try to find a counterexample to a hypothesis. Weights will be generated randomly once again. Lastly, we use pre-defined weights in instead of random weights to learn a smaller selection of models.

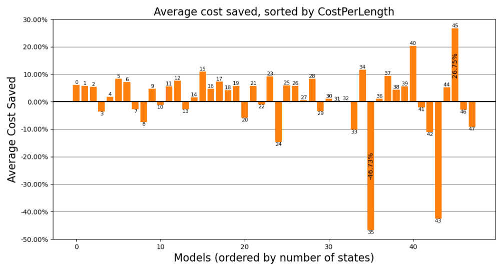
Average cost saved sorting pre-generated sequences



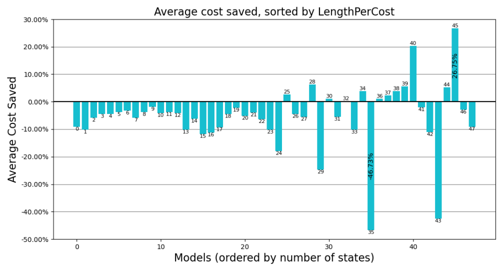
(a) Sorted by length



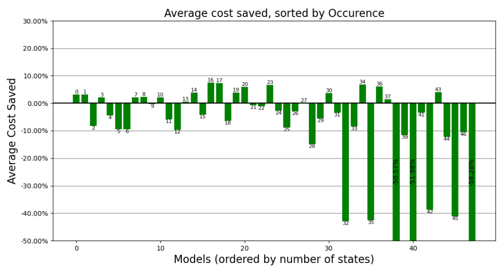
(b) Sorted by cost



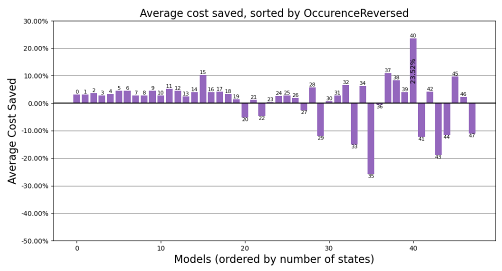
(c) Sorted by cost per length



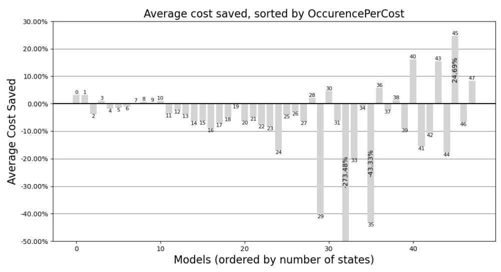
(d) Sorted by length per cost



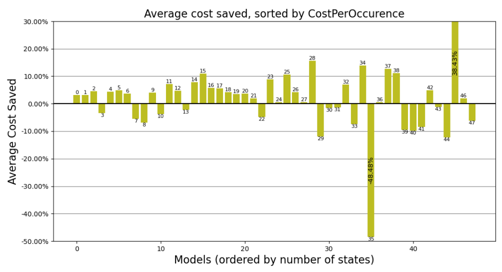
(e) Sorted by occurrence



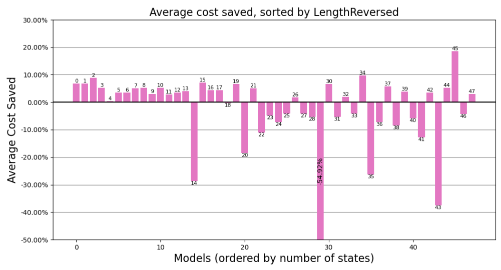
(f) Sorted by occurrence, reversed



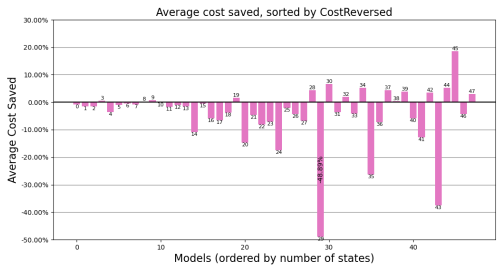
(g) Sorted by occurrence per cost



(h) Sorted by cost per occurrence



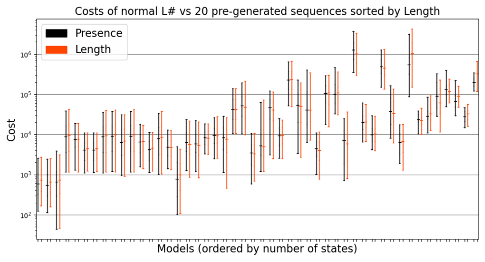
(i) Sorted by length, reversed



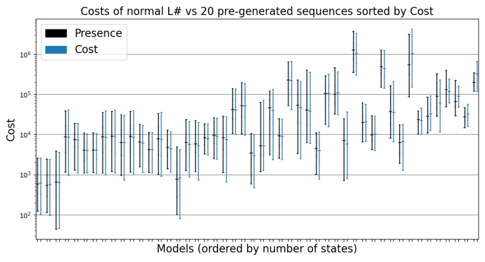
(j) Sorted by cost, reversed

Figure 24: Average cost saved with various ways of sorting 20 pre-generated test sequences.

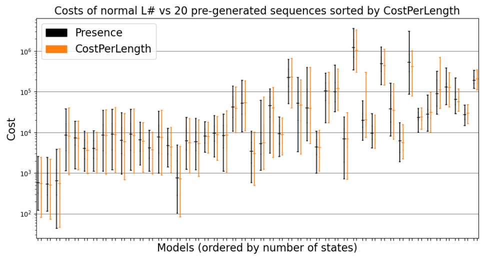
Average cost saved sorting pre-generated sequences



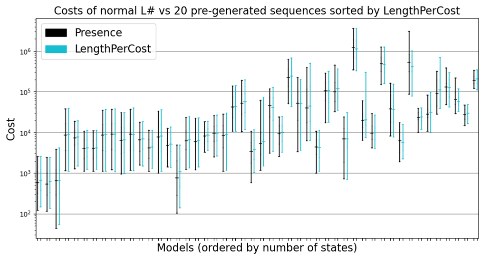
(a) Sorted by length



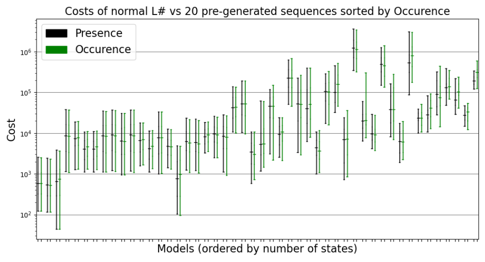
(b) Sorted by cost



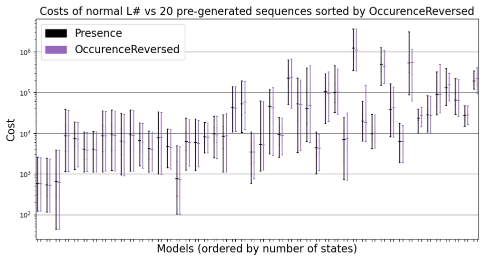
(c) Sorted by cost per length



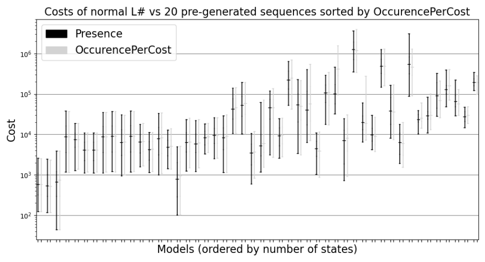
(d) Sorted by length per cost



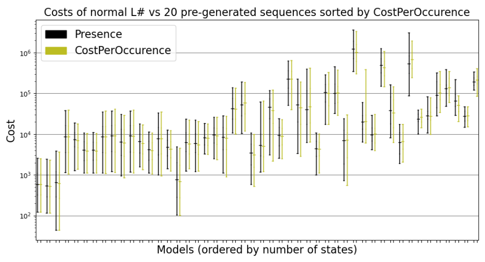
(e) Sorted by occurrence



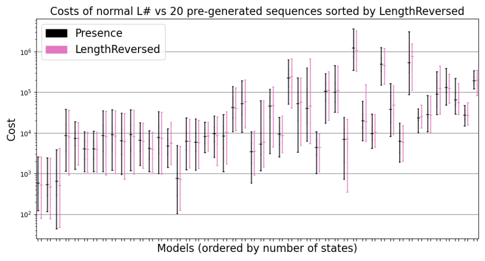
(f) Sorted by occurrence, reversed



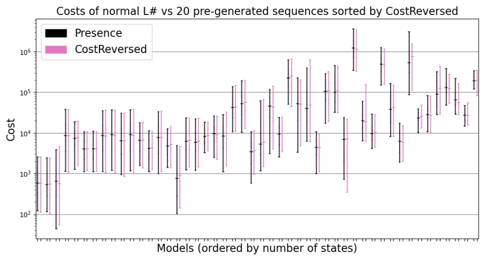
(g) Sorted by occurrence per cost



(h) Sorted by cost per occurrence



(i) Sorted by length, reversed



(j) Sorted by cost, reversed

Figure 25: Testing cost saved with various ways of sorting 20 pre-generated test sequences.

4.6.1 Randomly Generated Weights

Figure 24b shows the average cost saved by generating twenty test sequences and sorting them using each method compared with normal L# where only one test sequence is generated at a time. Bars above the line represent cost saved, bars below represent extra cost incurred.

Graphs 24a and 24b show the average cost saved by sorting the generated sequences by length and cost respectively. Both show very similar results, especially with the larger models on the left. For medium models, it does seem that sorting by cost saves cost more often than not.

For smaller models, sorting by cost per length (Subfigure 24c) seems to be generally result in cheaper runs. The inverse, sorting by length per cost (Subfigure 24d) tends to increase cost for most models. Curiously, one model benefits equally from cost per length and length per cost.

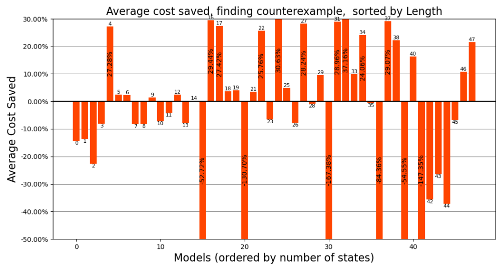
Sorting by occurrence seems to increase or at least not decrease cost for most models. Especially for larger models, where costs increased by 40% to 50% for most models. The opposite however, occurrence reversed, generally decreases cost a bit, but not as much as sorting by cost or length. More models benefit from sorting by reversed occurrence, than cost per length, like models 5, 6 and 7. Sorting by occurrence (Subfigure 24e) per cost generally increases cost, except for a few large models. Furthermore, it nearly tripled test phase cost for one specific model. Sorting by cost per occurrence (Subfigure 24f) is comparatively a much better idea. All small models tested saved cost by using this sorting method. Even some medium and large models benefitted, although some medium and large models had a decent increase in cost as well.

The graph of cost per occurrence (Subfigure 24h) looks similar to cost per length (Subfigure 24c).

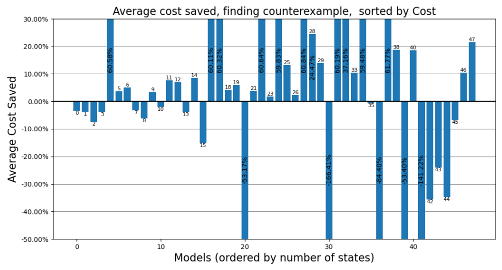
Length reversed (Subfigure 24i) does decent on smaller models. Likely because long sequences can teach a lot of information of a large part of the model early in the learning process. However, on larger models it increases cost instead, where the cost of each test sequence starts to add up. Sorting by cost reversed (Subfigure 24j) generally costs you more. An expensive sequence is not more likely to be a counterexample, thus more sequences that just cost more are tried.

One outlier in most graphs is the model number 46, which corresponds to 'model3.dot'. Almost all methods of sorting decreased cost spent, even seemingly contradictory ones like cost per length and length per cost.

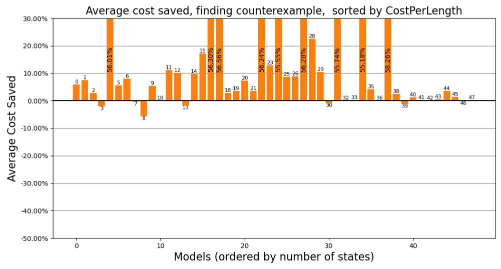
Average cost saved sorting pre-generated sequences finding counterexample



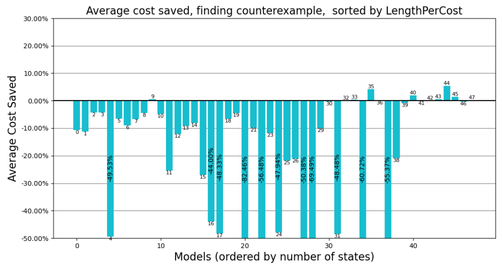
(a) Sorted by length



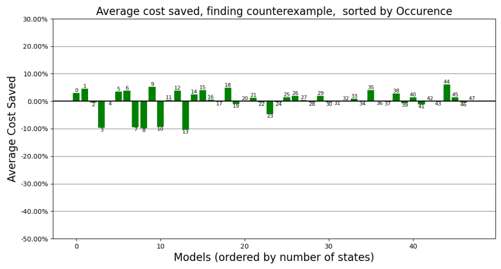
(b) Sorted by cost



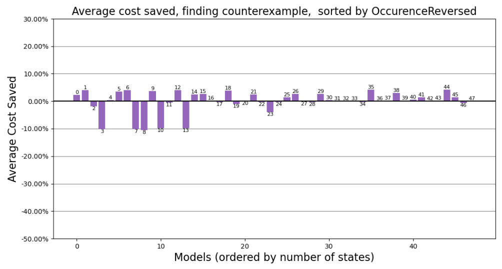
(c) Sorted by cost per length



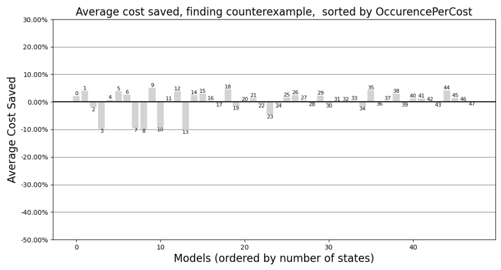
(d) Sorted by length per cost



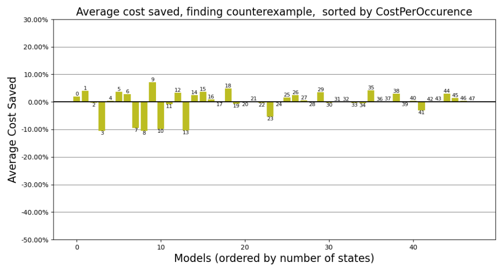
(e) Sorted by occurrence



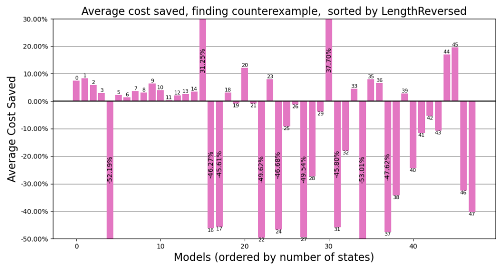
(f) Sorted by occurrence, reversed



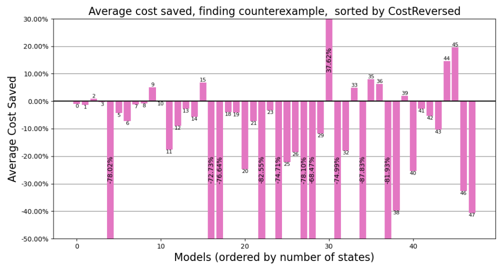
(g) Sorted by occurrence per cost



(h) Sorted by cost per occurrence



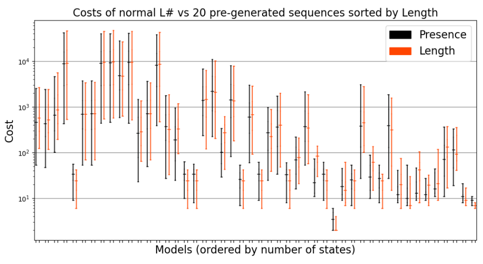
(i) Sorted by length, reversed



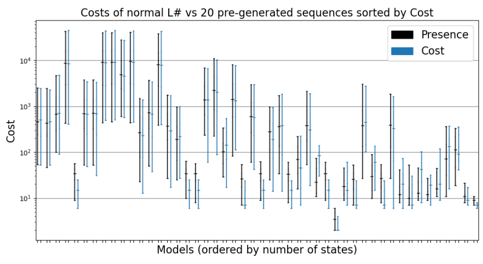
(j) Sorted by cost, reversed

Figure 26: Average cost saved finding a counterexample with various ways of sorting 20 pre-generated test sequences.

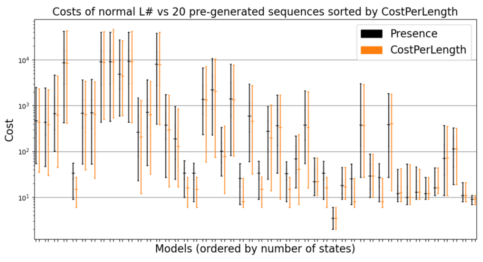
Cost saved sorting pre-generated sequences finding counterexample



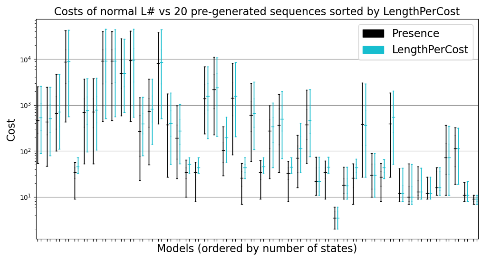
(a) Sorted by length



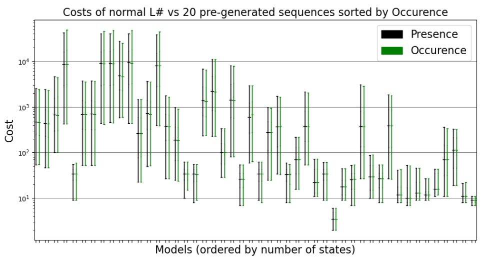
(b) Sorted by cost



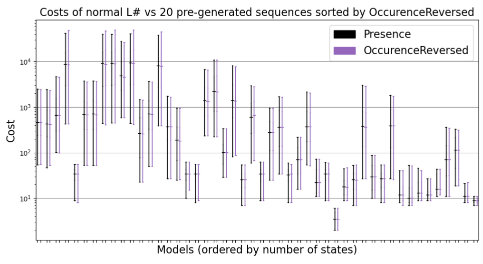
(c) Sorted by cost per length



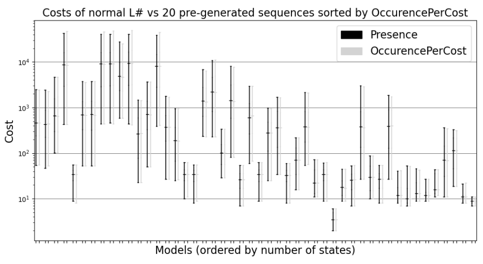
(d) Sorted by length per cost



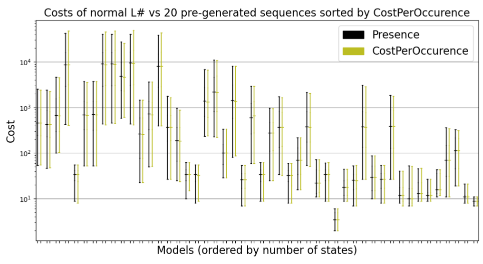
(e) Sorted by occurrence



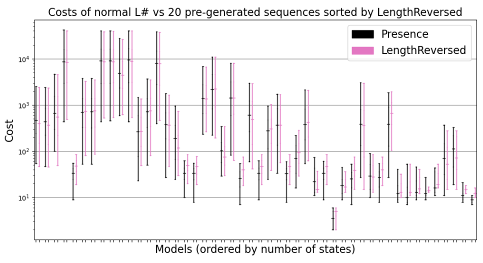
(f) Sorted by occurrence, reversed



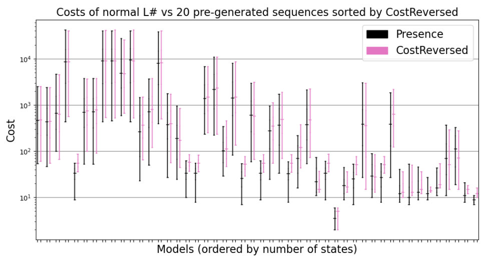
(g) Sorted by occurrence per cost



(h) Sorted by cost per occurrence



(i) Sorted by length, reversed



(j) Sorted by cost, reversed

Figure 27: Box-plot of costs saved finding a counterexample with various ways of sorting 20 pre-generated test sequences.

4.6.2 Counterexample Searching

Figure 26 shows average costs saved with the various sorting methods while finding a single counterexample. Compared to results in Figure 24 we see that cost savings and losses are more extreme.

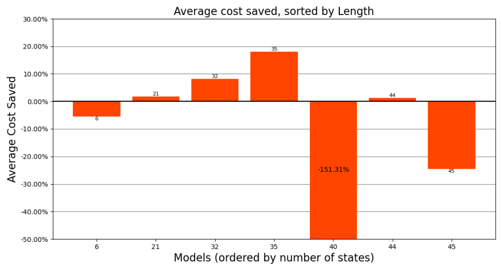
Sorting by cost (Figure 26b) has some big decreases in cost, but also some big increases for select models. Here, weighted L# always takes the cheapest test sequences first. This approach saves a decent amount of costs, but for some models the cheapest sequences are not very likely to be the counterexamples, which increases cost a lot.

Sorting by cost per length (Figure 26c) is generally a good idea to save costs, even more so than during the full learning process. Even for model 35, where the full process had 46.12% extra costs, now has a small decrease in costs.

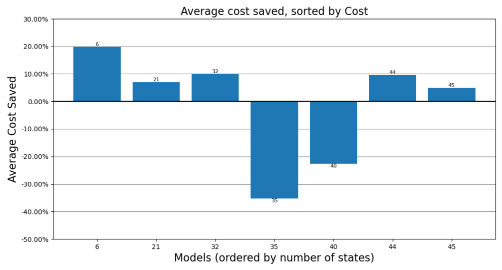
All sorting variants using occurrence are barely different from each other, they are also very close to the results of generating a single test sequence. This is because weighted L# did not have data on which inputs were in previous counterexamples, making it impossible to calculate an accurate occurrence value for each input. Instead, each occurrence value was set to zero as it would be at the start of the normal learning process.

Length reversed and cost reversed generally increase costs even when finding a counterexample. This should not be too surprising, as long sequences and expensive sequences are more expensive. Interesting exception is model 30, where costs decreased. This was because a long counterexample was needed, so sorting by cost or length increased the chance of finding this test sequence.

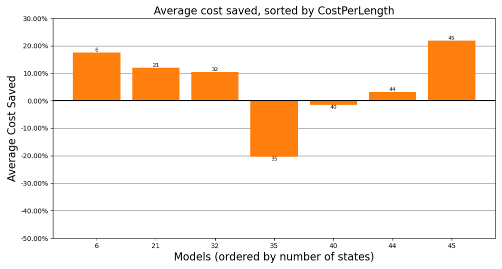
Average cost saved sorting pre-generated sequences with pre determined weights



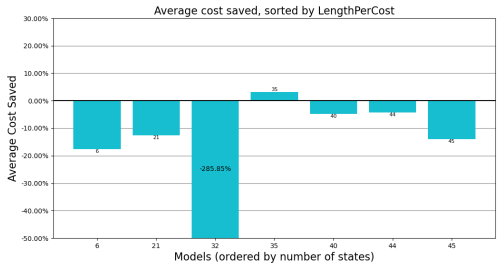
(a) Sorted by length



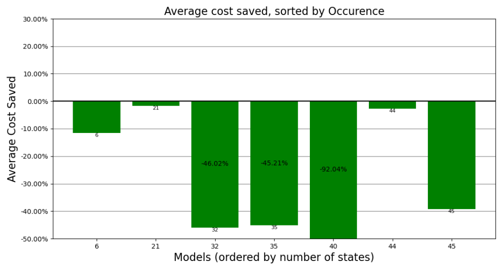
(b) Sorted by cost



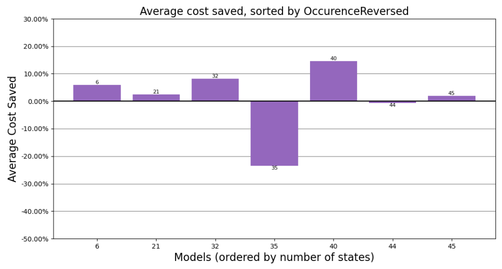
(c) Sorted by cost per length



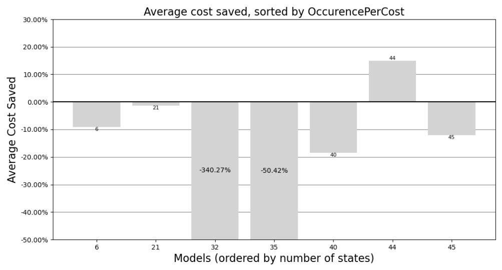
(d) Sorted by length per cost



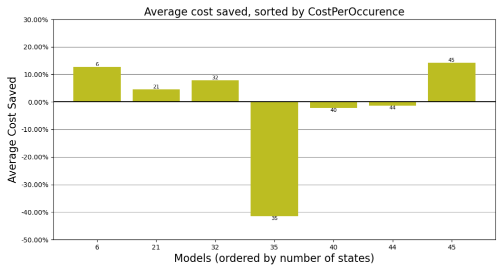
(e) Sorted by occurrence



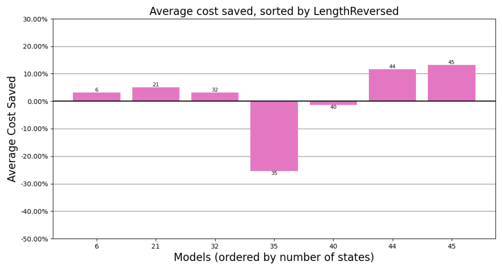
(f) Sorted by occurrence, reversed



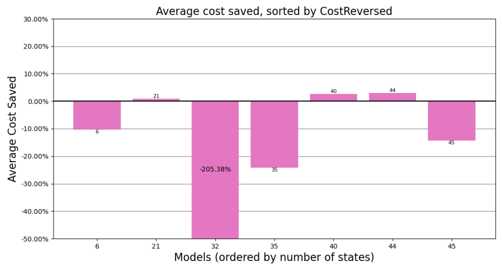
(g) Sorted by occurrence per cost



(h) Sorted by cost per occurrence



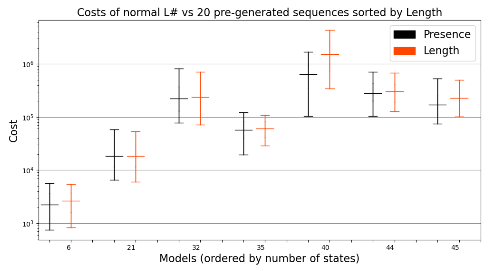
(i) Sorted by length, reversed



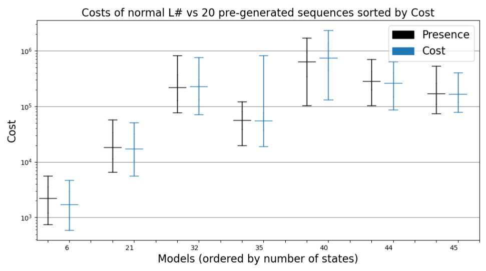
(j) Sorted by cost, reversed

Figure 28: Average cost saved with various ways of sorting 20 pre-generated test sequences, with pre determined weights

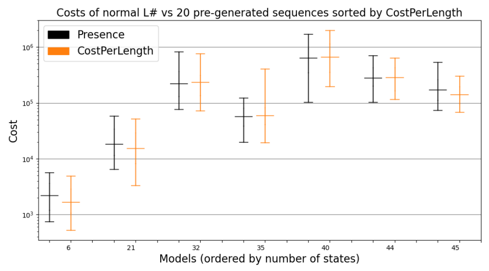
Cost saved sorting pre-generated sequences finding counterexample



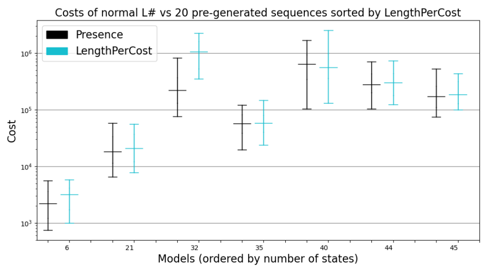
(a) Sorted by length



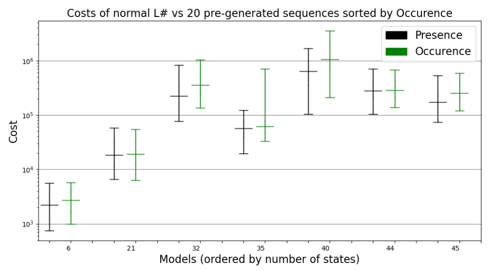
(b) Sorted by cost



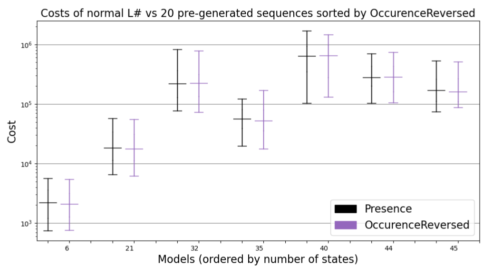
(c) Sorted by cost per length



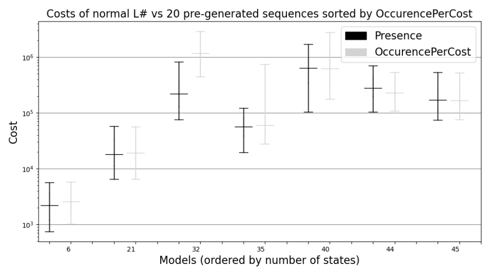
(d) Sorted by length per cost



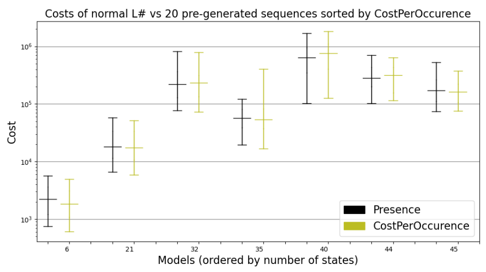
(e) Sorted by occurrence



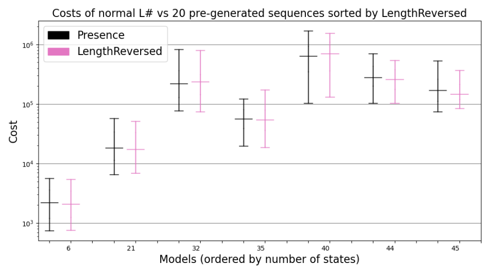
(f) Sorted by occurrence, reversed



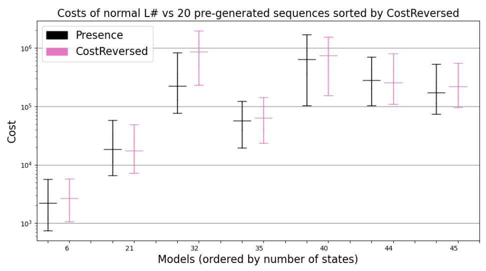
(g) Sorted by occurrence per cost



(h) Sorted by cost per occurrence



(i) Sorted by length, reversed



(j) Sorted by cost, reversed

Figure 29: Testing cost saved with various ways of sorting 20 pre-generated test sequences.

4.6.3 Predetermined Weights

Having pre-defined weights shows the effect of the sorting method when input costs where chosen in way that could make sense.

For 8 out of 10 sorting methods, we see that model 35, 'OpenSSH_2_2_2', has an increase in cost compared to normal L#. In Figure 28d it is the only one that has a decreased cost. This could mean that this model does not benefit from this measure in general.

Cost per Length made learning the OpenSSL, ASML and OpenSSH models cheaper, but the TCP models more expensive. Exception to this is model 40, 'OpenSSH.dot', which turned significantly more expensive. While the random weight full runs found a positive effect of the measure, with this weight, we found a negative effect instead.

Sorting by cost was only bad for model 35, likely for a similar reason as why the weighted random measure was bad. Having two expensive inputs in row in a test sequence makes it less likely for it to end up in the top of the list of test sequences. This does not cause a ballooning of costs like with the weighted random measure. Because with a list of test sequences, normal L# will still eventually try those sequences even if they are expensive.

The sorting method with the most change compared to the random weight runs is Length per Cost, which favors short sequences with high costs. With sensible weights this method is almost always increases cost. This graph now shows that using this sorting method on models 32, 40, 44 and 45 increases cost, compared to the random weights, where the method saved cost instead. The one exception to this is model 35, where this method decreased costs instead. Most likely because fully learning this model often involves a relatively short counterexample with two expensive inputs in a row.

5 Discussion

R1 How effective is a test sequence with a cost-minimal P at reducing learning cost?

There definitely exist cases where using Dijkstra to generate the access sequences decreases the cost of learning certain models. When states can be reached with multiple test sequences, choosing the cheapest access sequence generally saves costs compared to choosing the shortest access sequence found with BFS. This comes at the cost of test sequence diversity, which can cause the learning algorithm to miss counterexamples for a long time. Counterexamples it would have had the opportunity to learn from without the cheapest access sequence. An example where this problem could surface is when a model branches in two similar submodels, by way of two different inputs with different costs. One of those inputs will never be chosen as part of the access sequence, which means that the learning algorithm cannot distinguish this part of the model from the one accessed with the cheaper input.

R2 How effective is a test sequence with a weighted I at reducing learning cost?

Using a weighted random distribution to pick inputs instead of a normal uniform random distribution is a more risky choice than using Dijkstra for the access sequence instead of BFS. The way this measure generates a random walk can dramatically decrease the chance of expensive inputs being in the random walk to begin with. If these expensive inputs are behind important transitions in the model, then those transitions will only be found when you are very lucky. A use case for this measure is when it is very clear that the expensive inputs are not important to the functioning of the model you are trying to learn. This could be the case for models with 'error' inputs of which is known that they go to a sink state.

R3 How effective is a test sequence with a state-pairwise cost-minimal W at reducing learning cost.?

Compared to Dijkstra and the weighted random walk, it is much harder to say whether minimising the separating sequence is an effective way to reduce learning costs for a specific type of model. We do see that the Cartesian-Dijkstra method does not scale well. From the experiments we do see that as model size increases, the cost of using Cartesian-Dijkstra increases as well. Using a separating sequence of minimal cost for every state-pair increases the amount of separating sequences per state (unless you are lucky and one can function for two or more at once), meaning that the chance of picking the separating sequence that distinguishes the hypothesised state from the actual state gets smaller, which increases the average amount of tests that need to be done to find a counterexample. However, from the experiment results we see that the decrease in cost is worth it for smaller models, but the scale tips to traditional methods for larger models.

R4 How effective is a test sequence with a cost-minimal P , a weighted I and a state-pairwise cost-minimal W at reducing learning cost?

For the most part, combining all test sequence cost minimising measures combines all aspects of each measure. For most models this means greatly decreases performance, because one of the three measures greatly decreased performance. However, with intentional weights we still see that performance could increase. When trying to learn a model, carefully choosing weights and which measures to pick could save costs compared to using one measure or none at all.

R5 How effective is sorting a pre-generated list of test sequence at reducing learning cost?

There are models where the sorted pre-generated list of test sequences can decrease cost with certain sorting methods. Of note are sorting by cost, cost per length, occurrence-reversed and length-reversed.

Sorting by cost decreased learning costs, but when learning a model that requires a test sequence that includes an expensive input it can instead be more expensive to learn. Sorting by cost biases toward shorter sequences, removing this bias by sorting with cost per length seems to be beneficial for some models, with an increased performance compared to sorting by cost itself. This aspect makes it easier to use this sorting method as a default choice, when not much is known about a model. Sorting by cost per length was the best method when looking for a counterexample to a hypothesis. It saved costs for most models tested or was only marginally more expensive.

Sorting by length-reversed can make learning cheaper for models where counterexamples are long, like in large models, or where long sequences can help with finding more states early, like in small models. Occurrence-reversed showed a small decrease in cost for most models.

Recommendation As shown above, there is no measure that guarantees you save cost that you can use on any SUL with any weights. However, with a little prior knowledge, you can be decently sure that certain measures can save you cost. When the only thing you know about your SUL is its inputs and their costs, and similar inputs have similar costs, then using Dijkstra to generate the p has a good chance to save some costs. When you also know that your SUL has a low amount of states (below 10 states), then you can use Cartesian-Dijkstra to make a W set instead of Hybrid-ADS or the Wp-method to save some costs as well. When you are really sure that all the expensive inputs only go to a sink state and are not used for anything else, then a weighted random i will save a lot of costs. The option of going through a pre-generated list of test sequences sorted by cost per length generally saves costs, but when it does not save cost, then the extra costs are not as much as other sorting methods.

6 Conclusion

This thesis presents various measures of potentially reducing the cost of model learning when SUL inputs have different costs. Several measures to reduce testing phase cost are discussed. First we showed how to reduce cost for each part of the PIW test sequence. We generated a minimal cost P by using Dijkstra. We reduced the cost of I by using a weighted random distribution based on the input cost instead of a uniform distribution. We presented a way to generate a minimal cost distinguishing sequence for every state-pair with 'Cartesian-Dijkstra' for W.

Another method is to create a pre-generated list of test sequences and when searching for a counterexample, going through that list from cheapest to most expensive, as well as other orders including: length, cost per length or occurrence: how often the inputs where used in previous counterexamples. All these measures were implemented within a fork of the L# Learning Library.

We did experiments testing the effectiveness of the different measures, measuring how much cost was or was not saved by using L# with and without measures. The experiments showed that none of the measures saved costs in all situations. That said, some measures saved cost in most situations, while some saved a lot in very particular cases.

As shown above, there is no measure that guarantees you save cost that you can use on any SUL with any configuration of weights. However, with a little prior knowledge, you can be decently sure that certain measures can save you cost. When the only thing you know about your SUL is its inputs and their costs, and similar inputs have similar costs, then using Dijkstra to generate the p has a good chance to save some costs. When you also know that your SUL has a low amount of states (below 10 states), then you can use Cartesian-Dijkstra to make a W set instead of Hybrid-ADS or the Wp-method to save some costs as well. When you are really sure that all the expensive inputs only go to a sink state and are not used for anything else, then a weighted random i will save a lot of costs. The option of going through a pre-generated list of test sequences sorted by cost per length generally saves costs, but when it does not save cost, then the extra costs are not as much as other sorting methods. When you know that a model resembles a line with no branches, then sorting by occurrence can decrease cost as well.

6.1 Future Research

This thesis focused on reducing costs of inputs to the SUL, but the reset cost can be quite significant as well. Future research could look at ways of reducing resets, or ways of learning without the need to reset the SUL. A way to give the reset a cost was implemented for this thesis as well, but was unused.

This thesis did not look at methods of reducing costs outside the testing phase. One such method could be to give a number of cheap input sequences to the SUL before starting the learning process proper. This might give the algorithm a head start, but could also lead it the wrong way, so carefully choosing input-output might be the way to go.

An issue noted by this thesis is that prioritizing cheaper test sequences reduces the diversity of the kinds of inputs we give to the SUL. One way of reintroducing test sequence diversity is by alternating between L# test sequence generation with and without cost saving measures. Another way of reintroducing diversity is to change cost saving strategies partway through. Some cost saving strategies presented in this thesis worked well early in the learning process. You could start with these strategies at first and then move to different strategies or normal L# at the end.

Additionally, there are aspects of the measures that I did not have time to implement and/or test. First is the effect of the size of the pre-generated test sequence list. A longer list would mean more cheap sequences are tested before expensive sequences, but takes more memory and potentially important expensive sequences take longer to be tested. Regarding the method of generating the W-set, I did not have time to find a method that tries to find a middle ground between cheap distinguishing sequences and distinguishing sequences that covers as many state-pairs as possible.

Finally, the strategies in this thesis where only tested and measures based on theoretical costs. Future work could try these techniques in practice on a real SUL with real (time) costs.

References

[1] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6.

[2] Kousar Aslam et al. "Interface protocol inference to aid understanding legacy software components". In: *Softw. Syst. Model.* 19.6 (2020), pp. 1519–1540. DOI: 10.1007/S10270-020-00809-2.

[3] *Automata Wiki.* URL: https://automata.cs.ru.nl/.

[4] Nicolas Brémond and Roland Groz. "Case Studies in Learning Models and Testing Without Reset". In: *ICST Workshops.* IEEE, 2019, pp. 40–45. DOI: 10.1109/ICSTW.2019.00030.

[5] Tsun S. Chow. "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Trans. Software Eng.* 4.3 (1978), pp. 178–187. DOI: 10.1109/TSE.1978.231496.

[6] André Takeshi Endo and Adenilso da Silva Simão. "Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods". In: *Inf. Softw. Technol.* 55.6 (2013), pp. 1045–1062. DOI: 10.1016/J.INFSOF.2013.01.001.

[7] Susumu Fujiwara et al. "Test Selection Based on Finite State Models". In: *IEEE Trans. Software Eng.* 17.6 (1991), pp. 591–603. DOI: 10.1109/32.87284.

[8] Bharat Garhewal and Carlos Diego Nascimento Damasceno. "An Experimental Evaluation of Conformance Testing Techniques in Active Automata Learning". In: *MODELS.* IEEE, 2023, pp. 217–227. DOI: 10.1109/MODELS58315.2023.00012.

[9] Alex Groce, Doron A. Peled, and Mihalis Yannakakis. "Adaptive Model Checking". In: *Log. J. IGPL* 14.5 (2006), pp. 729–744. DOI: 10.1093/JIGPAL/JZL007.

[10] Loes Kruger, Sebastian Junges, and Jurriaan Rot. "Small Test Suites for Active Automata Learning". In: *CoRR* abs/2401.12703 (2024). DOI: 10.48550/ARXIV.2401.12703. arXiv: 2401.12703.

[11] *L# Learning Library Repository.* URL: https://gitlab.science.ru.nl/sws/lsharp.

[12] David Lee and Mihalis Yannakakis. "Principles and methods of testing finite state machines-a survey". In: *Proc. IEEE* 84.8 (1996), pp. 1090–1123. DOI: 10.1109/5.533956.

[13] David Lee and Mihalis Yannakakis. "Testing Finite-State Machines: State Identification and Verification". In: *IEEE Trans. Computers* 43.3 (1994), pp. 306–320. DOI: 10.1109/12.272431.

[14] Gang Luo, Alexandre Petrenko, and Gregor von Bochmann. "Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines". In: *Protocol Test Systems: 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems.* 1995, pp. 95–110. DOI: 10.1007/978-0-387-34883-4_6.

[15] Joshua Moerman. "Nominal Techniques and Black Box Testing for Automata Learning". PhD thesis. Radboud University, 2019, pp. 34–35.

[16] Edward F. Moore. "Gedanken-Experiments on Sequential Machines". In: *Automata Studies. (AM-34), Volume 34.* 1956, pp. 129–154. DOI: doi:10.1515/9781400882618-006. URL: https://doi.org/10.1515/9781400882618-006.

[17] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. "Black Box Checking". In: *FORTE.* Vol. 156. IFIP Conference Proceedings. Kluwer, 1999, pp. 225–240.

[18] Alexandre Petrenko et al. "Nondeterministic State Machines in Protocol Conformance Testing". In: *Protocol Test Systems.* Vol. C-19. IFIP Transactions. North-Holland, 1993, pp. 363–378.

[19] Ronald L. Rivest and Robert E. Schapire. "Inference of Finite Automata Using Homing Sequences". In: *Inf. Comput.* 103.2 (1993), pp. 299–347. DOI: 10.1006/INCO.1993.1021.

[20] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *USENIX Security Symposium.* USENIX Association, 2015, pp. 193–206.

[21] Wouter Smeenk et al. “Applying Automata Learning to Embedded Control Software”. In: *ICFEM*. Vol. 9407. Lecture Notes in Computer Science. Springer, 2015, pp. 67–83. DOI: 10.1007/978-3-319-25423-4_5.

[22] Martin Tappler, Bernhard K. Aichernig, and Roderick Bloem. “Model-Based Testing IoT Communication via Active Automata Learning”. In: *ICST*. IEEE Computer Society, 2017, pp. 276–287. DOI: 10.1109/ICST.2017.32.

[23] Frits W. Vaandrager. “Model learning”. In: *Commun. ACM* 60.2 (2017), pp. 86–95. DOI: 10.1145/2967606.

[24] Frits W. Vaandrager et al. “A New Approach for Active Automata Learning Based on Apartness”. In: *TACAS (1)*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 223–243. DOI: 10.1007/978-3-030-99524-9_12.

[25] M. P. Vasilevskii. “Failure diagnosis of automata”. In: *Cybernetics* 9.4 (1973), pp. 653–665. DOI: 10.1007/BF01068590.

Appendix

Nr.	Name	States	Inputs	Seeds	Weightfiles
0	4_learnresult_SecureCode_Aut_fix.dot	4	14	50	50
1	ASN_learnresult_SecureCode_Aut_fix.dot	4	14	50	50
2	1_learnresult_MasterCard_fix.dot	5	15	50	50
3	ASN_learnresult_MAESTRO_fix.dot	6	14	50	50
4	RSA_BSAFE_Java_6.1.1_server_regular.dot	6	8	50	50
5	OpenSSL_1.0.2_client_regular.dot	6	7	50	50
6	OpenSSL_1.0.1j_client_regular.dot	6	7	50	50
7	4_learnresult_MAESTRO_fix.dot	6	14	50	50
8	4_learnresult_PIN_fix.dot	6	14	50	50
9	Rabo_learnresult_SecureCode_Aut_fix.dot	6	15	50	50
10	Rabo_learnresult_MAESTRO_fix.dot	6	14	50	50
11	miTLS_0.1.3_server_regular.dot	6	8	50	50
12	OpenSSL_1.0.1l_client_regular.dot	6	7	50	50
13	10_learnresult_MasterCard_fix.dot	6	14	50	50
14	OpenSSL_1.0.2_server_regular.dot	7	7	50	50
15	Volksbank_learnresult_MAESTRO_fix.dot	7	14	50	50
16	GnuTLS_3.3.12_server_regular.dot	7	8	50	50
17	GnuTLS_3.3.12_client_regular.dot	7	8	50	50
18	NSS_3.17.4_client_regular.dot	7	8	50	50
19	NSS_3.17.4_server_regular.dot	8	8	50	50
20	RSA_BSAFE_C_4.0.4_server_regular.dot	9	8	50	50
21	OpenSSL_1.0.2_client_full.dot	9	10	50	50
22	GnuTLS_3.3.12_client_full.dot	9	12	50	50
23	learnresult_fix.dot	9	15	50	50
24	GnuTLS_3.3.12_server_full.dot	9	12	50	50
25	OpenSSL_1.0.1l_server_regular.dot	10	7	50	50
26	OpenSSL_1.0.1g_client_regular.dot	10	7	50	50
27	GnuTLS_3.3.8_client_regular.dot	11	8	50	50
28	OpenSSL_1.0.1j_server_regular.dot	11	7	50	50
29	NSS_3.17.4_client_full.dot	11	12	50	50
30	TCP_FreeBSD_Client.dot	12	10	50	50
31	GnuTLS_3.3.8_server_regular.dot	12	8	50	50
32	ASML_3_3_3_3.dot	13	8	50	50
33	TCP_Windows8_Client.dot	13	10	50	50
34	GnuTLS_3.3.8_client_full.dot	15	12	50	50
35	OpenSSH_2_2_2.dot	15	8	50	50
36	TCP_Linux_Client.dot	15	10	50	50
37	GnuTLS_3.3.8_server_full.dot	16	11	50	50
38	OpenSSL_1.0.1g_server_regular.dot	16	7	50	50
39	DropBear.dot	17	13	50	50
40	OpenSSH.dot	31	22	30	30
41	model4.dot	34	14	30	30
42	model1.dot	35	15	30	30
43	TCP_Windows8_Server.dot	38	13	30	30
44	TCP_FreeBSD_Server.dot	55	13	30	30
45	TCP_Linux_Server.dot	57	12	30	30
46	model3.dot	58	22	30	30
47	BitVise.dot	66	13	30	30

Table 2: Model Information