A Reinforcement-Learning Style Algorithm for Black Box Automata

Itay Cohen
Department of Computer Science
Bar Ilan University
Ramat Gan 52900, Israel
itay.cohen5@live.biu.ac.il

Roi Fogler
Department of Computer Science
Bar Ilan University
Ramat Gan 52900, Israel
roi.fogler@gmail.com

Doron Peled
Department of Computer Science
Bar Ilan University
Ramat Gan 52900, Israel
doron.peled@gmail.com

***Abstract*—The analysis of hardware and software systems is often applied to a *model* of a system rather than to the system itself. Obtaining a faithful model for a system may sometimes be a complex task. For learning the regular (finite automata) structure of a black box system, Angluin's L^* algorithm and its successors employ membership and equivalence queries. The regular positive-negative inference (RPNI) family of algorithms use a less powerful capability of collecting observations for learning, with no control on selecting the inputs. We suggest and study here an alternative approach for learning, which is based on calculating utility values, obtained as a discounted sum of rewards, in the style of reinforcement learning. The utility values are used to classify the observed input prefixes into different states, and then to construct the learned automaton structure. We show cases where this classification is not enough to separate the prefixes, and subsequently remedy the situation by exploring deeper than the current prefix: checking the consistency between descendants of the current prefix that are reached with the same sequence of inputs. We show the connection of this algorithm with the RPNI algorithm and compare between these two approaches experimentally.**

I. Introduction

Studying and analyzing models is one of the basic themes of computer science. Models can be used to study the very basic idea of computability (Turing Machines) and complexity. Models are also used to analyze systems, e.g., in testing and in model checking. Different models are used in understanding natural languages, and, on the other hand, in performing different tasks during compilation of programs. Often, the object to which one applies an algorithm is a model of an actual system, rather than the system itself. This allows the algorithm to become affordable, when the actual system is not directly accessible, or contains far too many details to be analyzed. Consequently, the results of using such a model become dependent on its faithfulness.

Software and hardware verification often uses a model that is created by the person/team that performs the verification. One can try to learn a model of a system through interacting with it, or even try to apply the verification algorithm *while* interacting directly with the system. Unfortunately, learning a model, while performing an algorithm, or as a preliminary step, is of high complexity. A prominent example for this situation is a "combination lock", which is a sequence of n elements, each one of which with k possibilities. Overall, there are k^n possibilities for the combination, and this is also the order of magnitude of experiments that are needed to learn it. On the other hand, the high complexity is an advantage when such a lock is used to protect one's valuable things. Unfortunately, a combination lock can be embedded in a learned model, even in a basic model like finite automaton.

Learning a finite automaton model, i.e., a *black box* automaton, has gained considerable attention in the research community. It can be part of natural language recognition [11], or a part of verifying a black box finite state system [21]. One notable algorithm is Angluin's L^* learning algorithm [2]. This algorithm uses two kinds of queries: (a) membership queries, that are used to check if a given sequence belongs to the learned language (i.e., is accepted by the corresponding regular automaton), and (b) equivalence queries, which are used to check if a conjectured automaton is indeed equivalent to the learned language, and if not, returning a distinguishing example. Implementing the equivalence learning is computationally hard, exactly because of the combination lock phenomenon: checking an exponential number of combinations, in the size of the learned automaton, is required to guarantee that such a combination does not lead to a behavior distinguishing the learned and the actual automata. The Angluin family of algorithms [2], [12], [13], [24] have control on the learned model that allows to query about any particular input and obtain the output.

Another family of algorithms [11] are based on obtaining a set of input sequences with the corresponding output; there is no capability of choosing the input, or comparing a candidate model. These algorithms provide a good match to the reported samples and in the limit, given a large enough number of reported experiments, can converge in the limit into an exact representation. The RPNI algorithm [19] [20] is a prominent example of such an algorithm, based on positive and negative samples.

We suggest here a *utility based* learning algorithm that uses some elements of reinforcement learning. It collects observations in the style of Monte Carlo experiments from the black box automaton that is subject to learning. Reinforcement learning calculates a utility value for each one of the involved

This research was funded in part by EU2020 grant FOCETA: Foundations for Continuous Engineering of Trustworthy Autonomy.

states. This value approximates the *expected discounted sum of rewards* on paths that start after a given prefix. The goal in reinforcement learning is to optimize a path that commences from each state to its successor; such a path is obtained by selecting to move each time to a successor state that has a maximal utility value. In our case, the calculation of the utility value, which is a *numeric signature* of the prefix, is used to separate the provided *observation* prefixes according to the *states* reached. We will henceforth also say informally that the numeric signatures *separate the states* (reached using the prefix). The transition relation can then be calculated from the experiments by observing how a state that is reached in an observed prefix transfers to another reached state by the occurrence of an input. We show via counterexamples that it is not always sufficient to provide utility value in the style of reinforcement learning to achieve a a complete separation of the states of a black box through such a numeric signature. Subsequently, we fix this problem and achieve a full separation by checking in more depth the consistency based on continuations of the prefixes.

We compare different versions of our algorithm with the RPNI algorithm based on various selected examples of different characteristics, with varying number of states.

II. Preliminaries

A. The Model

A finite automaton $\mathcal{A} = (S, \iota, \Sigma, \delta, F)$ has the following components:

- S is a finite set of *states*.
- $\iota \in S$ is the *initial state*.
- Σ is a finite set of *inputs*.
- $\delta : S \times \Sigma \mapsto S$ is the *transition function*.
- $F \subseteq S$ are the *accepting* states. We also denote F alternatively as an *output function* $F : S \mapsto \{0, 1\}$, where $F(s) = 1$ when s is accepting, and $F(s) = 0$ otherwise.

We define δ^* as the transitive closure of δ, i.e., $\delta^*(s, \epsilon) = s$ and $\delta^*(s, x.a) = \delta(\delta^*(s, x), a)$.

An *execution* or *observation* $\sigma = s_0 a_1 s_1 \dots s_n$ is an alternating sequence of states and actions, where $s_0 = \iota$, and for each $0 \leq i < n$, $\delta(s_i, a_{i+1}) = s_{i+1}$. In fact, it is sufficient to write $\sigma = a_1 a_2 \dots a_n$ and the corresponding states will follow from the definition of the automaton. We will also use the notation $o_0 a_1 o_1 a_2 o_2 \dots o_n$ to denote the alternating sequence of outputs and inputs (with $o_0 = F(\iota)$).

In a *Markov Chain*, the transitions from a state are selected according to some probabilistic distribution. Thus, in addition to the above we have a probabilistic distribution $p : S \times \Sigma \mapsto [0, 1]$ such that for each $s \in S$, the sum of $p(s, a)$, where $a \in \Sigma$ is 1. This means that from the state s, the action a is taken with probability $p(s, a)$.

When learning an automaton $\mathcal{A}$, we assume that our system is a black box, where we know only the input alphabet Σ and can reset it to its initial state. In order to learn the structure of the black box, we can observe executions. The different learning algorithms that we will discuss make different assumptions about the ability to observe the executions. The Angluin learning algorithm assumes that we can issue an execution that starts with the initial state based on the inputs and inspect its outputs. The RPNI algorithm can only observe the inputs and outputs of the executions (again that start from the initial state) provided by the learned system. In the latter case, the selection of actions from states are done according to some probabilistic distribution. Thus, the black box is, in fact, a Markov Chain. We do not necessarily know the distribution, and it is not in our control. We are also not interested to learn this distribution; we are only interested in obtaining the transition relation. However, it is important to assume that there is such a distribution that is kept fixed along the learning process. The distribution may be state dependant, but not time dependant. This prevents the situation where there are two sequences σ and σ' that reach the same state s, where the probability of executing some sequence ρ after σ is different from the probability of executing ρ after σ'.

A *trie* or a *prefix tree* is a search tree $T = (G, \Sigma, \delta, r, w)$, with a set of states S, a set of symbols (letters) Σ, a transition function δ, an initial root state r and a labeling $w : G \mapsto \Sigma^*$. The edges (g, a, g'), i.e. where $\delta(g, a) = g'$, are labeled with a symbol $a \in \Sigma$. Each node $g \in G$ in the trie is associated with a word $w(g)$ that is obtained from concatenating the symbols labeling the edges from the root r to g. The root r is labeled with the empty word ϵ. A trie can be used, e.g., to store a dictionary, where the entry for a word $\sigma = w(g)$ is stored at the node g. The complexity of reaching a word σ from the root, by following the symbols of σ labeling the edges from the root of the trie, is then the length of it $|\sigma|$.

B. The Angluin's Learning Algorithm

In [2], Angluin describes an algorithm L^* for learning the minimal deterministic automaton that corresponds to a given black box $\mathcal{A}$. Angluin's learning algorithm builds a candidate automaton $\mathcal{A}^*$ by making observations, called membership queries, on the system $\mathcal{A}$, i.e., invoking a procedure $test(v)$ that returns 1 if v is executable in the black box after a **reset** to its initial state, and 0 otherwise. Once the algorithm has obtained a candidate solution $\mathcal{A}^*$ that is consistent with all the observations run so far, it uses a procedure called equivalence query that checks whether $\mathcal{L}(\mathcal{A}^*) = \mathcal{L}(\mathcal{A})$. If not, the procedure gives a minimal sequence σ (the *discrepancy*) distinguishing $\mathcal{A}^*$ from $\mathcal{A}$. The learning algorithm then uses σ to refine the current solution. The equivalence queries use further membership queries, based on the Chow-Vasilevskii algorithm [8], [25].

To construct a candidate automaton, the algorithm keeps two sets of sequences: a prefix-closed set of *access sequences* $V \subseteq \Sigma^*$ and a suffix-closed set of *distinguishing sequences* $W \subseteq \Sigma^*$. Each sequence v in V corresponds to reaching a state of $\mathcal{A}^*$ by executing v from s_0. Different sequences may lead to the same state.

The algorithm keeps a table $T : (V \cup V.\Sigma) \times W \to \{0, 1\}$ such that for any $v \in V \cup V.\Sigma$ we have $T(v, w) = 1$ if and only if $vw \in \mathcal{L}(\mathcal{A})$.

We define an equivalence between the rows of the table $\sim \subseteq V.\Sigma \times V.\Sigma$ as $v \sim v'$ if $T(v,w) = T(v',w)$ for every $w \in W$. The access sequences in V will eventually, correspond to nodes in the candidate automaton constructed from the table T; then equivalent rows correspond to reaching the same state from the initial state with the row's access sequence. To construct the candidate automaton from the table T, it is necessary that T is *closed*, i.e., for every $v \in V$ and $a \in \Sigma$ there exists $v' \in V$ with $v' \sim va$. This means that the *successors* of access sequences in V are equivalent to some access sequences that are already in V. If T is not closed, we add va to V and fill the table with new rows $va.\Sigma$ and columns w according to membership queries. It is also necessary that the table T is *consistent*, i.e., for all $v \sim v'$, if $T(v,a) = 1$, then $T(va,w) = T(v'a,w)$ for all $w \in W$. Intuitively, this means that if two rows in the table correspond to the same state, then their corresponding successors also corresponding to the same state. If this is not the case, the sequence aw is added to W and we fill the table accordingly with membership queries.

When the table T is closed and consistent, we set $\mathcal{A}^* = ([V/\sim], \epsilon, \Sigma, \delta)$, where the transition relation δ is defined as follows. Let $[v]$ be a $\sim$ equivalence class of v. Set $\delta([v],a) = [v']$ when $v' \sim va$. This relation is well defined when the table T is closed and consistent. We then invoke an equivalence checking oracle on $\mathcal{A}^*$. If the oracle returns a discrepancy σ, for each prefix v of σ that is not in V, we add v to V and update T accordingly.

The L^* algorithm makes $O(n^2|\Sigma|)$ membership queries and at most n calls to the oracle. Its running time is $O(n^3|\Sigma|) + T_{oracle}$, where T_{oracle} is the total time spent by the oracle [22]. Unfortunately, the implementation of the oracle [8], [25], carries out a lower bound of exponential time complexity in the size of the learned automaton. The Angluin algorithm achieves an exact representation of the learned black box. However, it assumes control over the membership queries it uses. Furthermore, implementing the equivalence queries necessarily takes exponential time.

C. The RPNI Algorithm

The RPNI algorithm [19] [20] consists of two phases. In the first phase, we collect a set of input sequences from Σ^* with their corresponding outputs. The sequences are divided to two sets: sequences with output 0, called *negative* and are denoted by S^-, and sequences with output 1, called *positive* and are denoted by S^+. The algorithm builds a prefix tree acceptor $PTA(S^+)$ using paths starting from the initial state are the collected positive input sequences (S^+).

In the second phase, the prefix tree is folded into an automaton by iteratively combining states with *consistent* futures. The consistency between states, defined below, means that their outputs cannot be distinguished by any negative sequence based on the collected sequences. In this phase, the nodes are partitioned into *red*, *blue* and *white*. Red nodes are those that were already processed. Blue nodes are the immediate successors of the red nodes, and are the candidates for being merged with red nodes in cases of compatibility. The other nodes are white.

The second phase progresses while not all the nodes are red, starting with the root being the only red node. Then, a blue node s_1 is selected, and the algorithm attempts to merge it with some red node s_2. For such a merge to succeed, the algorithm checks consistency with the negative examples (S^-) by validating that each of the negative sequences collected during the first phase does not accept in the merged automaton. Finding such inconsistent successors $s_1{}'$ and $s_2{}'$ means that s_1 and s_2 cannot be merged. On the other hand, if there are no inconsistencies, corresponding successors s_1 and s_2 due to a commonly labeled path are merged as well.

Let $|S^+|$ and $|S^-|$ be the sum of the lengths of prefixes in S^+ and S^- respectively. The prefix tree acceptor (PTA) contains $O(|S^+|)$ states. Hence, the loop described in the second phase performs $O(|S^+|)^2$ state merges. In addition, each merge can cause $O(|S^+|)$ merges in the worst case. The consistency check complexity is $O(|S^-|)$. Overall, the time complexity of the algorithm is $(O(|S^+| + O(|S^-|)) \cdot O(|S^+|^2)$

D. Reinforcement Learning

Reinforcement Learning (RL) includes methods for learning to efficiently interact with an environment [17]. The typical model for the environment in RL is a Markov Decision Process (MDP). At each step, an *action* is selected, and the successor state is reached depending on a probabilistic distribution that depends on the current state and action taken. While the selection of action is in the control of the system, and is the choice sought after by RL, the transition to the successor state is made by the environment, based on the corresponding probability distribution. A numeric *reward* is given to every selected action, and its value depends on the current state and the action taken.

The goal of reinforcement learning is to find an optimal selection among the possible choices when interacting with the environment. Optimality is defined as maximizing a utility value assigned to each state of the environment related to the fitness of the future selections and based on the future rewards. Often, this value is the expected sum of rewards associated each with an action selected during the execution. In order not to consider infinite sums, the values along the execution can be discounted by γ^n with respect to the future distance n from the current point, with $0 < \gamma \leq 1$ (or, alternatively, be summed up with respect to a finite horizon).

RL calculates a *strategy* $\pi : S \mapsto T$ that selects the action that needs to be taken from a given state s. The value $V_\pi(s)$ is the expectation (the results of the transitions can depend on the probabilistic distribution of the selected actions) of the discounted sum of rewards, starting from s, and selecting actions according to the strategy π. RL seeks a strategy that will maximize this utility value; the value of which is denoted as $V(s)$ without indicating the strategy in the subscript, as a unique "best" strategy is selected. RL algorithms can iterate between calculating the utility value and

improving the strategy, or can combine these two tasks into one [17].

A key aspect in finding an optimal strategy π that maximizes the values of $V_\pi(s)$ for each state s, is to estimate the utility values of a given strategy. When the probabilistic distribution p of transiting between states is known, one can directly calculate the utility values according to the Bellman Equation

$$V_\pi(s) = r(s, \pi(s)) + \gamma \times \Sigma_{s \in S}(p(s, \pi(s), s') \times V_\pi(s'))$$

by solving a set of equations, or using a fixpoint improvement calculations until it converges. When this distribution is not known, different methods are applied to estimate the utility values of a given strategy. One method is based on *Monte Carlo* experiments, where sums of rewards of sequences from s are sampled. For more information on RL techniques, see [17].

III. Utility Based Learning

A. The Basic Algorithm

We will describe now a black box learning algorithm that combines ideas from both Angluin's L^* algorithm, RPNI and reinforcement learning. The principles are as follows:

We do not know the states of the learned black box $\mathcal{B}$. Hence, we use a prefix tree to *reach* its different states. As in Angluin's algorithm, where the rows of the algorithm consist of a prefix tree, or in RPNI, there can be different sequences from the initial state that reach the same state and we need to separate these sequences. Angluin's algorithm uses distinguishing sequences, which consist of the columns in the table, to separate these sequences according to the different states in the learned automaton. RPNI combines states provided that their future successors in the constructed prefix tree are consistent with the negative observations. This means that if there is a suffix of the collected observations extending the current path from the root, which is marked with σ from both s and s' that disagree on acceptance, then s and s' cannot be combined.

The separation in our algorithm is based on a utility value $V(\sigma)$ assigned to a sequence σ in the generated prefix tree. The goal is to have in the limit that $V(\sigma) = V(\sigma')$ if and only if $\delta^*(\iota, \sigma) = \delta^*(\iota, \sigma')$. That is, the sequences σ and σ' reach the same state. We show later that we achieve this goal only partially, i.e., given enough observations, the *only if* part holds up to some minor difference (as calculation does not continue indefinitely), but not necessarily the *if* part. We later show how to overcome this problem, separating prefixes further by looking at the consistency of values between their corresponding extensions.

As in RL, this value is the expected discounted sum of rewards on paths that start at a given state in the prefix tree. The individual rewards values on a path depend on both the next encountered action, and, for additional flexibility,[1] whether or not the current and successor states are accepting or not. The expected value is calculated using RL Monte Carlo techniques, i.e., by calculating the discounted sum of rewards based on given observations of the black box. Observations start at the initial state ι of the black box, and the utility values are kept in a generated prefix tree. Intuitively, in an experiment $\rho.\delta$, we update the expected value after ρ based on the discounted sum of rewards calculated for δ.

In our case, the model to be learned, playing the role of an environment, behaves as a deterministic finite automaton rather than an MDP. In fact, unlike classical RL, we only have an environment that we want to learn, and do not have an agent that we want to control to interact with the environment. Thus, there is only one "strategy" and we directly calculate $V(\sigma)$ that depends on parameters of the environment rather than $V_\pi(\sigma)$. The system provides a set of observations that consist of inputs and outputs. There *can* be a probabilistic distribution where the system chooses some $a \in \Sigma$ when reaching its internal state s, as in a Markov Chain, but we are not aware of this probability, nor do we care to learn it.

Our goal is to estimate the utility values of the single strategy mentioned above. We do that by applying a Monte Carlo based technique. A reported sequence σ is an alternating sequence of outputs and inputs $o_0 i_1 o_1 i_2 \dots o_n$, where $i_j \in \Sigma$ and $o_k \in \{0, 1\}$. We calculate the *discounted sum* $ds(\sigma)$ using the following parameters:

- γ The discounting factor.
- $R(i)$ The reward for getting out of an accepting state by taking an action i.
- $R'(i)$ The reward for getting into an accepting state by taking an action i.
- $r(o_j, i_j, o_{j+1})$ The reward assigned to the symbol i_j, based on its adjacent outputs o_j, o_{j+1}. It is defined as follows:

$$r(o_j, i_j, o_{j+1}) = \begin{cases} R(i_j) & \text{if } o_j = 1 \\ R'(i_j) & \text{if } o_j = 0 \text{ and } o_{j+1} = 1 \\ 0 & \text{otherwise} \end{cases}$$

Then, ds is defined recursively as follows:

- $ds(o) = 0$ (This is the case of a sequence of size 0 with only a single output o).
- $ds(o_j i_j o_{j+1} \rho) = r(o_j, i_j, o_{j+1}) + \gamma \times ds(o_{j+1} \rho)$.

Note that the reward does not depend on the current state, as we do not know the state of the learned automaton.

We keep a data structure $V(\sigma)$ that returns the calculated value for the expected discounted sum for the continuation of any collected prefix σ. The structure can be organized as a trie (a dictionary tree) or using a hash table. In fact, we need to use as a key only the sequence of inputs from σ, and the outputs are redundant. There are two elements that are kept: $V.E(\sigma)$, which is the partially calculated expected discounted sum itself, and $V.N(\sigma)$, which is the number of continuations considered after σ. The longer σ is, the less accurate its expected sum becomes. Hence, we keep the elements $V.E, V.N$ for prefixes up to a certain length. This length is denoted by acc_len. The update for a sequence $\sigma.\rho$ is defined using the procedure $update(\sigma, \rho, acc\_len)$, where σ

[1] This in fact guarantees that we will separate at least the prefixes that end with an accepting state from those that end with a non-accepting state.

is the prefix that was already processed; it can consist of only the inputs, and ρ is the suffix that is needed to be considered.

Algorithm 1 Monte Carlo Utility Value Estimation

```
procedure UPDATE(σ, ρ, acc_len)
    switch ρ do
        case ρ = o
            ds := 0 (end of sequence)
        case ρ = o_j i_j o_{j+1} ρ'
            ds := r(o_j, i_j, o_{j+1}) +
                  γ × update(σ o_j i_j, o_{j+1} ρ')
    if |σ| ≤ acc_len then
        if new(σ) then
            V.N(σ) := 0
        end if
        V.E(σ) := V.E(σ) × V.N(σ)/(V.N(σ)+1) + ds/(V.N(σ)+1)
        V.N(σ) := V.N(σ) + 1
    end if
    return ds
end procedure
```

Note that the selection of the given observations are not in the control of the learning algorithm. We do not assume a particular probability distribution on the selection of the actions (inputs) from any given state; nor we are interested to learn these probabilities, but merely the structure of the automaton. With different probabilities, we will obtain different values.

In the limit, the utility value $V(\sigma)$ and $V(\sigma')$ are the same, when the prefixes σ and σ' in the prefix tree end with the same state. Of course, the values calculated by our algorithm for $V(\sigma)$ and $V(\sigma')$ only approximate the limit values. We cluster the prefixes over their utility values using the DBSCAN clustering algorithm [16]. We fix the parameters for this algorithm: the minimal number of points required to form a dense region $minPts$ is set to 1. Another parameter that we fix is the density parameter ϵ, which controls how far joint values can be apart from each other while considered to be in the same cluster.

After clustering, the transition relation of the learned automaton can now be easily recovered from the prefix tree, since a prefix that ends with a state s and is extended by an action a to a state s' forms a transition $\delta(s,a) = s'$. The initial state ι is identified with the state at the root r of the prefix tree. Accepting states can be identified from the prefix tree as those that have output 1.

The complexity of DBSCAN is $\mathcal{O}(n^2)$ in the number of points to be clustered [26]. But the average time complexity is $\Theta(n \times \log(n))$ (when the ϵ parameter is chosen in a meaningful way) [16]. In our case, n is the size of the prefix tree.

B. A Limitation of the Utility Based Separation

The presented approach of assigning rewards may suffice to separate the states from one another in a variety of finite automata. However, in some cases, it is impossible to distinguish between two automaton states solely by their utility values based on the expected sum of future rewards as defined above. We show a simple example that demonstrates that a one dimensional numeric signature for each prefix might be insufficient to learn an automaton.

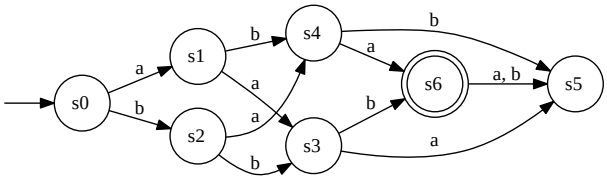

Fig. 1. This automaton cannot be learned based on numeric signatures alone

In figure 1, the finite automaton accepts only four words over the alphabet $\Sigma = \{a, b\}$. To prove that this automaton cannot be learned using the proposed approach, we need to show that two or more automaton states cannot be distinguished, for any parameters choice of the model. We focus on states s_1 and s_2 and prove that based on the current model, $V(s_1) = V(s_2)$ (since in the limit the value assigned to a prefix σ depends only on its last state s, we denote here $V(s)$ instead of $V(\sigma)$) for any choice of parameters. We assume we cannot control the transition selection probabilities of selecting a after the state s in the black box, denoted as $p(s,a)$. Therefore it is sufficient to prove that there exists an transition selection distribution for which two or more automaton states cannot be distinguished.

For the sake of the proof, assume that the transitions of the black box were drawn from a uniform distribution $p(s,a) = 1/|\Sigma|$ for each $s \in S$. Recall that we do not have control over the distribution, hence it is sufficient to fix a particular case of a distribution for which the selection of the (in fact any) reward parameters would not help making the separation. Let F be the accepting states of the automaton, and δ be its transition function. Again, we denote the rewards for getting out of an accepting state by $R(a), R(b)$ and $R'(a), R'(a)$ for getting into an accepting state. In any other case we set the reward to be 0. The reward function is defined as follows:

$$r(s,a) = \begin{cases} R(a) & \text{if } s \in F \\ R'(a) & \text{if } s \notin F \text{ and } \delta(s,a) \in F \\ 0 & \text{otherwise} \end{cases}$$

At this point, we can directly evaluate the utility values of the different states using the Bellman Equation. Let γ be the discount factor and s be a given automaton state.

$$V(s) = \sum_{a \in \Sigma} p(s,a)(R(s,a) + \gamma V(\delta(s,a)))$$

We then construct a linear system of equations for our example, based on the above formula.

$$\begin{bmatrix} 1 & -\frac{\gamma}{2} & -\frac{\gamma}{2} & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -\frac{\gamma}{2} & -\frac{\gamma}{2} & 0 & 0 \\ 0 & 0 & 1 & -\frac{\gamma}{2} & -\frac{\gamma}{2} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -\frac{\gamma}{2} & -\frac{\gamma}{2} \\ 0 & 0 & 0 & 0 & 1 & -\frac{\gamma}{2} & -\frac{\gamma}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 & -\gamma \\ 0 & 0 & 0 & 0 & 0 & 0 & 1-\gamma \end{bmatrix} \times \begin{bmatrix} V(s_0) \\ V(s_1) \\ V(s_2) \\ V(s_3) \\ V(s_4) \\ V(s_5) \\ V(s_6) \end{bmatrix} =$$
$$\begin{bmatrix} 0 & 0 & 0 & \frac{R'_a}{2} & \frac{R'_b}{2} & \frac{R_a}{2}+\frac{R_b}{2} & 0 \end{bmatrix}^T$$

The solution for this linear system of equations is the utility values of each automaton state.

$$\begin{bmatrix} V(s_0) \\ V(s_1) \\ V(s_2) \\ V(s_3) \\ V(s_4) \\ V(s_5) \\ V(s_6) \end{bmatrix} = \frac{1}{4} \begin{bmatrix} \gamma^2 R'_a + \gamma^2 R'_b + \gamma^3 R_a + \gamma^3 R_b \\ \gamma R'_a + \gamma R'_b + \gamma^2 R_a + \gamma^2 R_b \\ \gamma R'_a + \gamma R'_b + \gamma^2 R_a + \gamma^2 R_b \\ 2R'_a + \gamma R_a + \gamma R_b \\ 2R'_b + \gamma R_a + \gamma R_b \\ 2R_a + 2R_b \\ 0 \end{bmatrix}$$
$$\Rightarrow V(s_1) = V(s_2).$$

Consequently, the utility based approach would identify s_1 and s_2 as a single state. It is easy to show that by selecting another distribution for the learned automaton, like $p(s,a) = 0.4$, $p(s,b) = 0.6$, separation *will* be achieved.

It is natural to ask whether we can refine the calculation of the utility value in order to be able to achieve the separation for any learned automaton. One proposed refinement is to set a discount factor γ_σ for each observed symbol $a \in \Sigma$, such that if $a \neq b$, then $\gamma_a \neq \gamma_b$. It is easy to show that this refinement allows learning the automaton presented in figure 1, assuming the same transition selection distribution as before.

Although the refined approach is richer than its previous version, we devised another example where it fails. In figure 2, we present an automaton that accepts only six words over the alphabet $\{a, b, c\}$. In this example, we focus on states q_1 and q_2. The accepting suffixes of q_1 are "abc", "cab", "bca", while the accepting suffixes of q_2 are "bac", "cba", "acb". By applying a similar analysis as before, it can be demonstrated that the states q_1 and q_2 have an identical expected sum of rewards, under the different discount factors assumption. In fact, this pair of states has the same utility value for any selection of probability distribution of black box transitions, and for any discount factors values assigned for each selected action.

Although a direct use of utility values cannot always ensure a full separation, it can be proved that some separation can be achieved. Based on the reward function defined in this section, it is easy to see that an accepting state and a nonaccepting state always have different utility values.

Lemma III.1. *Let s be a state such that $s \in F$ and s' be a state where $s' \notin F$. Then there exists a reward function r and a discount factor γ such that $V(s) \neq V(s')$.*

IV. Refining the Utility Based Learning

The mentioned counterexamples are the evidence that a full separation of states might be a challenging task when relying exclusively on the calculated utility values. Therefore, we designed a mixed approach with two variants. This approach uses the utility values to obtain a initial separation of observed prefixes. Then, this separation is refined by a further analysis on the prefixes that happen to be in the same cluster. This analysis is done in two similar ways as shown in the following subsections.

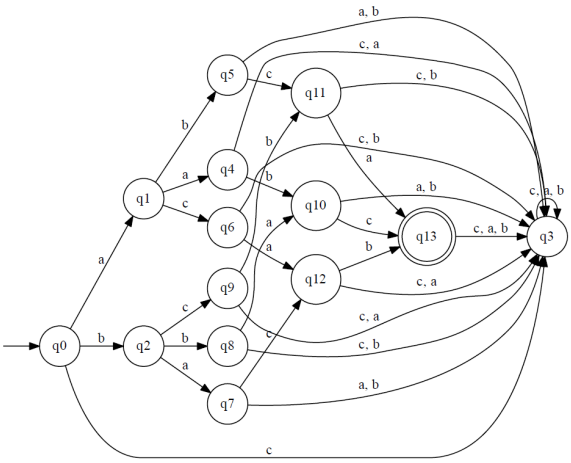

Fig. 2. Automaton that cannot be learned through refining the discount factor

A. Utility Based Learning with Lookahead

As discussed in section III, learning an automaton at the black box setting can be somewhat challenging when relying entirely on the approximated utility value of each prefix. Indeed, the expected sum of rewards for each prefix captures a portion of each state's characteristics, however we observed that there are cases where this information might be insufficient to separate all the states. To mitigate this, one possible approach is to obtain an initial partition of the observed prefixes using the utility value of each prefix, and then refine this partition by further analyzing prefixes that were clustered together. Considering prefixes σ, σ' that were associated to the same state by the initial partition. We try to separate σ from σ' by checking if there exists a ρ such that $accept(\sigma.\rho) \neq accept(\sigma'.\rho)$, while $accept$ is a predicate relying on the *accept/reject* information collected for each observed prefix on the value approximation phase.

We now describe the course of the algorithm of this approach. We denote the sequence (episode) length in the Monte Carlo experiment by ep_len. Long prefixes are revisited less frequently than shorter ones, therefore their approximated utility values tend to be less accurate. Hence, prefixes that are longer than e.g., $0.65 \times ep\_len$, are not considered for the clustering that creates the initial partition between observed prefixes. Shorter prefixes ($< 0.65 \times ep\_len$) will be referred to as the *kernel* of the prefix tree containing all the observed prefixes.

As in the basic utility based approach, the initial separation of the kernel prefixes was accomplished using clustering over their utility values using DBSCAN. Then, we refine

the clustering; we inspect each cluster and attempt to split it to subclusters, if possible. For each cluster, we inspect its associated prefixes one by one; for each prefix ρ, we check if its descendants accept/reject information is consistent with each one of the already created subclusters. In case it is inconsistent with all of them, a new subcluster is created.

The refined clusters are considered to be the learned automaton states and we can construct the structure of the automaton (transition function, initial state and accepting states) using the prefix tree as described above for the basic utility based approach.

We assume that V is the utility value function of a given prefix. Let $accept$ be a predicate relying on the *accept/reject* information collected for each observed prefix. For two prefixes ρ_1 and ρ_2 we denote $\rho_1 \equiv \rho_2$ if they represent the same state of our black box automaton.

Lemma IV.1. *$\rho_1 \equiv \rho_2$ if and only if $V(\rho_1) = V(\rho_2)$ and for each suffix σ, $accept(\rho_1.\sigma) = accept(\rho_2.\sigma)$.*

B. Utility Based with Dual Clustering

The above approach presents a refinement of the initial separation, using an accept/reject information of the observed prefixes. We assume that beyond the *kernel*, the numeric signature may not be as accurate as inside the kernel. However, despite being less accurate, those values may still capture *some* relevant information for splitting the initial clusters correctly. We present here an approach that is based entirely on utility values to refine the initial separation. This approach lets us refine the initial separation based on applying clustering again, this time for longer prefixes. This allows us to learn the final partition of prefixes without relying on the accept/reject information.

This approach uses clustering twice in the following manner. At first, an initial partition of prefixes is obtained using clustering over the kernel prefixes. This time, we set $kernel\_len$ to be $0.5 \times ep\_len$. Let the density parameter of the first separation be ϵ_1.

We define a broader region of prefixes that will be clustered again – the *compatible region*. We denote the compatible region boundaries by $comp\_len$, and set $comp\_len = 0.75 \times ep\_len$. Prefixes shorter than $comp\_len$ are referred to as *compatible region prefixes*. Since this region contains some longer prefixes with less accurate utility values, it is assumed that a coarser clustering is obtained in this step. To obtain a coarser separation, a larger density parameter is used, denoted by ϵ_2.

Similarly to the previous approach, we now refine the kernel prefixes separation. However, this time the refinement is done by considering an extension of prefixes up to the compatible region boundaries. To split an initial cluster to subclusters we iterate its elements likewise. However, when checking consistency between prefix extensions, their coarse clustering association is considered, instead of their accept/reject information.

We discuss the convergence proof of the dual clustering approach. For simplicity, we assume that our alphabet contains two symbols, $\Sigma = \{a, b\}$. Denote the rewards for getting out of an accepting state by R_a, R_b, and $R_a^{'}, R_b^{'}$ for getting into accepting state. In any other case, the reward will be 0. We Assume that T is the prefix tree and V is the utility value function of a given prefix. For two prefixes ρ_1 and ρ_2 we denote $\rho_1 \equiv \rho_2$ if they represent the same state of our black box automaton.

Lemma IV.2. *$\rho_1 \equiv \rho_2$ if and only if $V(\rho_1) = V(\rho_2)$ and for each suffix σ, $V(\rho_1.\sigma) = V(\rho_2.\sigma)$.*

V. Experiments

We compare our reinforcement-learning-style approaches with the RPNI algorithm, taking into account the minimal amount of samples required for correct learning, and the average learning time. The utility based learning with lookahead approach requires collecting random episodes from a black box automaton. The RPNI algorithm learns from collecting positive and negative sets of samples, not necessarily observing acceptance and rejection of prefixes of these samples. To better compare between RPNI and our algorithm, we set a baseline setting to our experiments. Accordingly, both RPNI and our algorithm are required to gather their samples using the black box queries outlined above.

Suites of Examples: We experimented with five suites of examples, each suite consists of automata with a varying amount of states.

The first suite consists of four "combination lock" automata, where the smallest one has four states and the largest one has seven. These automata are challenging to learn, as all the accepting strings own a *single* prefix of the length of the automaton. For instance, figure 3 shows the largest automaton in this suite. All its accepting prefixes share the common prefix "$aaabba$".

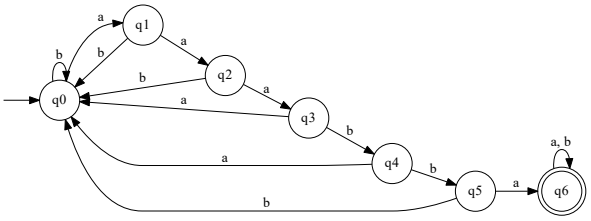

Fig. 3. A combination lock automaton with seven states

In section III we discussed an example that cannot be learned using our basic utility based approach. We devised a set of five automata based on this example with a varying number of states. We called them "cross" automata. Figure 4 shows one example in this set with eleven states. This set of automata might be somewhat challenging for our approaches to learn since its examples contain a few pairs of states with an identical expected sum of rewards. We have already proved that the automaton in figure 1 has one such pair, and it can be proved that a cross automaton with fifteen states has five of these pairs.

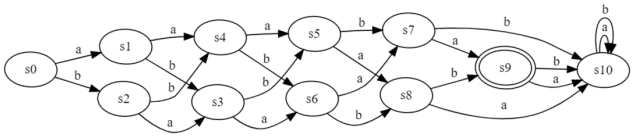

Fig. 4. A cross automaton with eleven states

The third suite of examples are automata that validate a correct structure of balanced parentheses. We devised a set of four automata with different lengths. The smallest of them (figure 5) accepts balanced parentheses with three open parentheses at the most. The largest automaton accepts balanced parentheses with six open parentheses at most. This suite of examples is somewhat challenging to learn, since very few generated sequences are accepted.

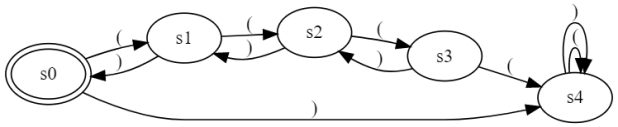

Fig. 5. Automaton that validates a structure with three balanced parentheses at most

The fourth suite of examples consists of automata that represent divisibility machines. With only two possible transitions out of each state, we check binary numbers divisibility. We assume the bits are scanned left to right as the way they are normally written. Thus, the most significant bit (msb) is scanned first and the least significant bit (lsb) is scanned last.

Let n be the value of a binary number at some point in the scanning process and let its remainder upon division by d be r. Hence, n can be written as $n = md+r$ for some integer m. Possible values for the remainder are $r \in \{0, 1, \ldots, d-1\}$. These remainder values will correspond to the states of the automaton. If b is the value of the next scanned bit (lsb) then the new value of the binary number will be $2n+b$ and the new remainder will be $(2r+b) mod\ d$. The automaton then proceeds to the new remainder state. We start in $r = 0$ state and if a sequence ends there, the number it represents is divisible by d. Our examples consists of divisibility machines by 4 (Figure 6), 5, 6, 7 and 9. Note that in these examples, every prefix can be extended into an accepting one.

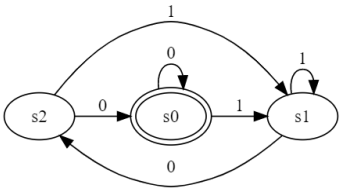

Fig. 6. Divisibility by 4 automaton

The last suite of examples are four automata that represent a combination of two simple divisibility machines. Let $Div(m, n)$ be the language $\{w \mid \#_a(w) \mod m = 0 \text{ and } \#_b(w) \mod n = 0\}$ over the alphabet $\{a, b\}^*$. The automata in this suite accept the languages $Div(2, 2), Div(2, 3), Div(2, 4), Div(3, 3)$. Figure 7 exhibits the automaton that accepts the language $Div(2, 3)$.

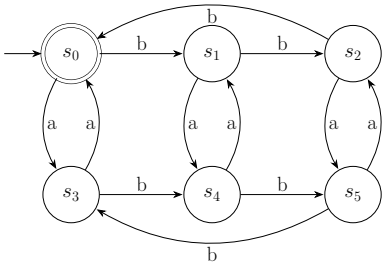

Fig. 7. Automaton that accepts the language $Div(2, 3)$

RPNI and Utility Based Learning with Lookahead Comparison: We first compare the utility based learning with lookahead approach with the RPNI algorithm in the black box automaton setting in terms of time and minimal amount of samples required for learning. The results of processing time and number of collected samples for both algorithms are the average results of ten tests with a success rate of at least 80%. In each one of the two algorithms we checked and used the black box episode length that was minimal for a successful learning.

To make a proper comparison, we also implemented the RPNI algorithm, following [19]. The implementation can be divided to two major parts, building the Prefix Tree Acceptor (PTA), and the learning part. To build the PTA, we applied the same method as used in our implementation to generate episodes that are valid sequences under the given alphabet. Each prefix of a generated episode was added to a positive set or a negative set according to its accept/reject information. A PTA and its transitions were updated according to the positive samples. In addition, the accepting states were marked as accepting in the PTA. After building the PTA, a minimal automaton was created, according to the following scheme: a "blue" state was selected and tried to merge with "red" state. The merge occurs if the resulted automaton is consistent with all the negative samples.

All the experiments were conducted on the same machine with an Intel® Core™ i7 CPU at 2.6 GHz and 16GB of RAM, running on a Windows 11 operating system. Both algorithms were implemented in Python. The source code for our implementation is available on GitHub [10].

We present here a description of the course of the experiments of the comparison. For each suite of examples, the rewards and discount factor values remained fixed for the utility based learning with lookahead approach. The only parameters that changed between different examples were the length of sampled episodes and the DBSCAN's ϵ value. We chose the minimal episode length that enabled a correct learning and adjusted the ϵ parameter to minimize the number of samples required for learning.

The first suite of examples we tested was the cross automata suite. These automata have a finite amount of accepting prefixes, which makes it easier to distinguish between automaton

Exp. (No. States)	Min. Amount of Samples		Learning Time (s)	
	UB learning	RPNI	UB learning	RPNI
Cross (7)	397	192	0.008	0.04
Cross (9)	555	416	0.008	0.12
Cross (11)	1454	419	0.09	0.27
Cross (13)	3228	522	0.12	0.64
Cross (15)	4089	620	0.2	0.93
Comb. Lock (4)	181	92	0.01	0.02
Comb. Lock (5)	591	210	0.01	0.06
Comb. Lock (6)	2031	192	0.25	0.06
Comb. Lock (7)	8148	200	1.32	0.07
Divisibility. 4 (3)	85	45	0.004	0.009
Divisibility. 5 (5)	221	172	0.01	0.08
Divisibility. 6 (5)	179	143	0.004	0.04
Divisibility. 7 (7)	1264	364	0.06	0.31
Divisibility. 9 (9)	1344	791	0.07	1.32
Parenthesis. 3 (5)	905	219	0.04	0.03
Parenthesis. 4 (6)	5252	590	0.23	0.12
Parenthesis. 5 (7)	21107	1483	1.22	0.439
Parenthesis. 6 (8)	86482	4192	3.43	1.4
Div(2,2) (4)	91	27	0.003	0.01
Div(2,3) (6)	232	60	0.02	0.04
Div(2,4) (8)	876	112	0.06	0.11
Div(3,3) (9)	958	96	0.08	0.15

TABLE I
BENCHMARK RESULTS - RPNI AND UTILITY BASED LEARNING WITH LOOKAHEAD

states. Hence, relatively less iterations were required to learn the different automata. Larger differences between the various rewards had high impact on the learning time. The episode length values varied between 9 for the smallest automaton in the suite and 13 for the largest automaton in the suite. When compared to RPNI, it is notable that utility based learning with lookahead needed *more* samples to successfully learn the different automata in this suite, but did it in less time.

For the combination lock suite, the rewards are chosen to be larger than the rewards in the previous suite. This is motivated by a few states having similar utility values in early experiments. Larger rewards helped in separating the values of these states. The episode length values were 8 for the smallest automaton in the suite and 13 for the largest automaton in the suite.

For the third suite, balanced parentheses, the examples are characterized by a relatively high ratio of negative/positive examples and as a result require longer episode lengths than the combination lock suite. Again, episode length and ϵ varied between the different examples. Depending on the size of the automaton, the length values varied between 10 to 19.

For the fourth suite, divisibility machines, the ratio of negative/positive examples is lower when compared to the previous suites. The reason behind this is that every prefix is extendable into an accepting sequence. Here, a relatively short episode length was required for a successful learning, 7 for the smallest automaton in the suite and 11 for the largest automaton. Consequently, learning these examples required a relatively short processing time. In comparison to RPNI, utility based learning with lookahead required more samples to successfully learn the automata in this suite, but did it in less time.

The last suite to be tested was the combined divisibility suite. Each automaton in this suite has a unique property - given a random prefix over the alphabet $\Sigma = \{a, b\}$, there in an identical probability for each automaton state to be associated with this prefix. As a result, the automaton state visits distribution is almost uniform in the Monte Carlo step. This enables the different utility values be relatively accurate within less iterations. The episode length values varied between 7 to 10. We can see that utility based learning with lookahead needed *less* time to successfully learn the different automata in this suite, but needed more samples to do so comparing to RPNI.

For each suite of examples, the rewards and the discount factors were identical across all examples. In table II, we exhibit the rewards given for each action in three different cases: getting out of an accepting state, entering an accepting case, and any other case. Different rewards values were given for different actions (values are separated by ',' in table). In addition, the discount factors we used were 0.6 for the cross, combination lock and combined divisibility suites and 0.2 for the divisibility and parenthesis suites.

Experiment	Reward acc-in	Reward acc-out	Reward rej
Cross	20,120	300,240	0,0
Comb. Lock	20,400	1200,800	0,0
Divisibility	20,120	300,240	0,0
Parenthesis	20,400	1200,800	0,0
Div(m,n)	20,120	300,240	0,0

TABLE II
REWARDS CHART - UTILITY BASED LEARNING WITH LOOKAHEAD

In the evaluation of RPNI across the different suites, the episode length was the only parameter to vary, i.e, there was no need for the parameter ϵ as DBSCAN was not used. It was selected to be the minimal successful length in term of learning the automata. The length values range between $8-16$ for the cross suite, $5-8$ for the combination lock suite, $4-8$ for the divisibility suite, $7-17$ for the parenthesis suite and $5-7$ for the combined divisibility suite. Full comparison between RPNI and utility based learning with lookahead is given in table I.

Dual Clustering Approach: We compare the dual clustering approach with RPNI. Similarly to the former approach, this approach requires a greater number of samples for learning an exact representation comparing to RPNI. As can be seen in table III below, in terms of processing time, this approach has faster processing time for two suites of examples out of five. Still, the results are slightly worse than the utility based learning with lookahead approach.

Experiment (No. States)	DC time (s)	RPNI time (s)
Cross (7)	0.02	0.04
Cross (9)	0.03	0.12
Cross (11)	0.25	0.27
Cross (13)	0.55	0.64
Cross (15)	1.21	0.93
Divisibility. 4 (3)	0.004	0.009
Divisibility. 5 (5)	0.01	0.08
Divisibility. 6 (5)	0.008	0.04
Divisibility. 7 (7)	0.1	0.31
Divisibility. 9 (9)	0.07	1.32

TABLE III
BENCHMARK RESULTS - RPNI AND DUAL CLUSTERING LEARNING TIME

Since dual clustering approach applies clustering twice, it requires an additional parameter, ϵ_2, which is the density parameter of the course clustering. In addition, learning an exact representation with this approach requires a relatively longer episodes. Hence, the episode length values were modified accordingly. Due to space limitations, we omitted the parameters table and full benchmark results for this approach.

Relying Exclusively on the Accept/Reject Information: When experimenting with the utility based learning with lookahead approach, it was evident that the accept/reject information of the observed prefixes was key in learning the right separation. To test effectiveness we use utility values that create the initial separation of kernel prefixes. We designed a second variant to the utility based learning with lookahead approach, in which the step of the initial separation based on utility values is completely omitted from the algorithm. The initial separation becomes a single cluster consists of all the observed prefixes. This cluster needs to be splitted solely based on the accept/reject information of the observed prefixes. Note that in this variant the rewards, discount factor and ϵ do not exist. We tested this variant on the different suites. Table IV exhibits the minimum samples and processing time for cross automata.

Experiment (No. States)	Samples	Time (s)
Cross (7)	1064	0.06
Cross (9)	1631	0.09
Cross (11)	4109	0.27
Cross (13)	8388	0.49
Cross (15)	13817	0.75

TABLE IV
RELYING EXCLUSIVELY ON THE ACCEPT/REJECT INFORMATION - CROSS AUTOMATA RESULTS

It is evident that in this suite, the utility values that create the initial separation between prefixes are indeed useful. The utility based learning with lookahead approach uses less samples to successfully learn the black box automaton, and as a result does it in a shorter amount of time.

VI. CONCLUSION

Obtaining a faithful model of the dynamics (i.e., time changeable) structure of a system has been the subject of vast research. Depending on the given information about the learned system, such algorithms may considerably differ from each other. Angluin's L^* algorithm for learning an automaton and its successors assume the capability of making experiments where inputs for the system are selected by the algorithm and the corresponding outputs are given. Moreover, the algorithm can use an oracle that compares between a candidate model and the actual system and provide a counterexample experiment in case they differ. On the other hand, the RPNI algorithm builds a model that is consistent with a collection of observations, without being able to control the selection of inputs in the observations or using an oracle to compare the model with the actual system.

We look here at learning a finite state model for a system based on collecting the same observations as in the RPNI algorithm. Both methods use a prefix closed set of examples. However, we calculate a utility value $V(\sigma)$ for prefixes σ of the collected observations. These values represent the expected value of the discounted sum of rewards, based on the sequence of inputs and outputs of that appear after σ in the collected observations. As such, the utility values depend on multiple future sequences occurring after σ. Our initial motivation is to separate prefixes σ and σ' that reach different states so that $V(\sigma) \neq V(\sigma')$. Otherwise, when σ and σ' end with the same state, $V(\sigma) = V(\sigma')$. Then, the states of the learned automaton can be obtained based on the separation of the prefixes. Furthermore, the transitions of the learned structure can be directly obtained based on the progress between states that is available in the collected observations. The values $V(\sigma)$ are used to separate states reached after observed prefixes. This can be comapared with the compatibility test between nodes to be unified in the RPNI algorithm, and using the separation sequences in the Anguluin algorithm.

It follows from the congruence property of automata theory that prefixes that reach the same state will have utility values that will become closer together as we collect more experiments. The first question that we ask is whether such a separation always exists. When choosing a fixed probability distribution of transitions, we show a negative example, where a few states will not be separated. This happens for *any* collection of rewards, which depends on both the inputs and the outputs (for generality) of each step in the observation. It is important, for a more refined version of our algorithm, that it is easy to guarantee that at least the sequences reaching the accepting states will be separated from the sequences reaching the non accepting states. Later on, we show that a more refined utility based approach, where different discounting factors are assigned to different selected actions may be sufficient to successfully learn the negative example. We then show a second counterexample, where this refined approach fails as well. In this example, *any* probability distribution of black box transition, and *any* assignment of discount factor values will not be able to provide separation. This case demonstrates that a utility based approach alone might be inadequate for the purpose of finite automata learning.

In order to rectify this problem, we suggested a mixed approach, where an initial separation is first achieved, and then this separation is refined by looking deeper into observations that extend the prefixes that were not distinguished. Accordingly, if $V(\sigma) = V(\sigma')$, then we will still try to separate σ from σ' by checking if there exists a ρ such that $V(\sigma.\rho) \neq V(\sigma'.\rho)$. Intuitively, ρ is used as a separation sequence, between σ and σ', which is an ingredient that exists implicitly in RPNI and explicitly in Angluin's algorithm. Alternatively, we can make such separation based only on the *accept/reject* information after each such sequence ρ. Consequently, we have experimented with several choices of our utility based approach and with RPNI.

Our experiments show mixed results; on the one hand, it seems that inherently, the utility value based approaches need relatively more samples to successfully learn an automaton compared to RPNI. On the other hand, in two automata suits

out of five, both utility value based approaches were faster when attempting to learn an exact representation of the black box automaton. The utility based learning with lookahead approach was faster in three suites out of five.

When considering the utility based learning with lookahead approach, we witnessed that the initial clustering based on utility values may assist in learning in a more efficient way comparing to solely relying on the accept/reject information of the observed prefixes. We showed this principle when experimenting with the cross automata suite. In addition, although utility based learning with lookahead needs more samples comparing to RPNI, sometimes it required *shorter* observations to correctly learn an automaton representation. For instance, in three out of five cross automata examples, RPNI could not learn a correct representation with the same length that was sufficient for utility based learning with lookahead (even when the RPNI was given all the possible prefixes up to this length). Intuitively, this stems from the additional information that is captured by the utility based approach, which depends also on the frequency of occurrences of observations; this can sometimes facilitate more accurate learning, achieving a finer separation.

When evaluating both utility value based approaches on the five automata suites, it is evident that the utility based learning with lookahead variant, which further refines the separation based on accept/reject outperformed the dual clustering approach in terms of processing time and amount of samples required for learning.

In general, we suggest that the RPNI algorithm should be more efficient for *hard cases*, e.g., when accepting executions are rare. It should require less experiments in order to achieve faithful learning, and in some but not all cases, the experiments are also shorter. This makes the stage of collecting observations often more efficient in RPNI. On the other hand, RPNI's complexity of the stage of folding the prefix tree into an automaton is cubic in the sum of lengths of the observations. This is a deficiency with respect to the utility based approach, where the complexity of creating an automaton from the prefix tree is quadratic in the size of the prefix tree, due to DBSCAN, and is often $\Theta(n \times log(n))$. As further work, we would like to perform additional experiments for establishing a more coherent comparison between the utility based approach and RPNI; in particular, looking at models that correspond to actual hardware and software systems.

Future work on this topic includes in evaluating the ability of our approach to achieve an *estimation* of an automaton, rather than achieving a completely faithful learning in comparison with other learning techniques (e.g., Angluin, RPNI). Another challenge is using the utility based approach for learning a probabilistic deterministic finite automaton, and comparison with algorithms such as Alergia [5], [14].

REFERENCES

[1] R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In *CAV'05*, LNCS, 2005.

[2] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75, 87-106 (1987).

[3] R. E. Bellman, Dynamic Programming, 1957.

[4] R. Alur, R. Grosu and M. McDougall. Efficient Reachability Analysis of Hierarchical Reactive Machines In *CAV'00*, LNCS 1855, p.280-295, 2000.

[5] R. .C. Carrasco, J. Oncina, Learning Stochastic Regular grammars by means of a state merging method. In Grammatical Inference AND Applications, 139–152, Springer-Verlag, 1994.

[6] E. M. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, 1999.

[7] E. Clarke, D. Long, K. McMillan. Compositional Model Checking. In *LICS'89*, IEEE , p.353-362, 1989.

[8] T.S. Chow. Testing software design modeled by finite-states machines. In *IEEE transactions on software engineering*, SE-4, 1978, 178-187.

[9] J. Cobleigh, D. Giannakopoulou, C. Pasareanu. Learning Assumptions for Compositional Verification. In *TACAS'03*, LNCS 2619, p.331-346, 2003.

[10] https://github.com/itay99988/Utility-Based-Learning.

[11] E. M. Gold, Language Identification in the Limit. Inf. Control. 10(5): 447-474 (1967).

[12] Malt. Isberner, F. Howar, B. Steffen, The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. RV 2014: 307-322.

[13] M.J. Kearns, U.V. Vasirani, An Introduction to Computational Learning Theory, MIT Press, 1997.

[14] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen, Learning Probabilistic Automata for Model Checking, Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, IEEE Computer Society, 2011, 111–120.

[15] A. Mazurkiewicz, Trace Semantics, Proceedings of Advances in Petri Nets, 1986, Bad Honnef, Lecture Notes in Computer Science, Springer Verlag, 279–324, 1987.

[16] M. Ester, H. Kriegel, J. Sander, X. Xu A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD'96*, LNCS, p.226-231, 1996.

[17] R. S. Sutton, A. G. Barto, Reinforcement Learning : An Introduction, MIT Press.

[18] E. Ochmanski, Languages and Automata, in The Book of Traces, V. Diekert, G. Rozenberg (eds.), World Scientific, 167–204.

[19] J. Oncina, P. García inferring regular languages in polynomial updated time, Series in Machine Perception and Artificial Intelligence, 1992, pp. 49–61.

[20] J. Oncina, P. García Identifying regular languages in polynomial time, Advances In Structural And Syntactic Pattern Recognition, , 1992, pp. 99–108.

[21] D. Peled, M. Vardi and M. Yannakakis. Black Box Checking. In *FORTE/PSTV'99*, 1999.

[22] R. Rivest and R. Schapire. Inference of Finite Automata Using Homing Sequences. *Information and Computation*, 103(2), p.299-347, 1993.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot. Duchesnay, E. Scikit-learn: Machine Learning in Python. *Journal Of Machine Learning Research*, p.2825-2830, 2011.

[24] F. W. Vaandrager, B Garhewal, J .Rot, T. Wißmann, A New Approach for Active Automata Learning Based on Apartness. 223-243

[25] M.P. Vasilevskii, Failure diagnosis of automata, *Kibertetika*, no 4, p.98-108, 1973.

[26] P. Viswanath and R. Pinkesh. l-dbscan : A Fast Hybrid Density based Clustering Method, Proceedings of the 18th Intl. Conf. on Pattern Recognition (ICPR-06), volume 1, pages 912.915, Hong Kong, 2006. IEEE Computer Society.

[27] W. Weimer and G. Necula Mining Temporal Specifications for Error Detection. In *TACAS'05*, LNCS 3440, p.461-476, 2005.

[28] G. Xie and Z. Dang. Testing Systems of Concurrent Black-boxes - an Automata-Theoretic and Decompositional Approach. In *FATES'05*, LNCS, 2005.