Universidad ORT Uruguay
Facultad de Ingeniería

Analysis, Evaluation and Improvement of Active Regular Inference Algorithms for Neural Sequence Acceptors

Entregado como requisito para la obtención del título de Ingeniería en Sistemas

Alejo Garat - 219610
Juan Pedro da Silva - 229475
Martín Iturbide - 241107

Tutores: Sergio Yovine y Franz Mayr

2024

Declaración de Autoría

Nosotros, Alejo Garat, Juan Pedro da Silva y Martín Iturbide declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el Proyecto Final de Ingeniería en Sistemas;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

[illegible]

Martín Iturbide

[illegible]

Juan Pedro da Silva

[illegible]

Alejo Garat

21-03-2024

Agradecimientos

Dedicado a nuestras familias, quienes nos han acompañado en esta etapa y fueron nuestros soporte.

Agradecemos a nuestros tutores, Dr. Sergio Yovine y Mag. Franz Mayr, por las largas discusiones que dieron fruto a este trabajo.

Abstract Español

El presente trabajo contribuye al campo de la Inteligencia Artificial Explicativa (XAI, por sus siglas en inglés). Nuestro objetivo es mejorar algoritmos, optimizar procesos y realizar una revisión exhaustiva del estado del arte. En concreto, se proponen optimizaciones para los algoritmos de extracción, probados en la competencia internacional TAYSIR, y proponemos la implementación del algoritmo Observation Pack con el fin de mejorar la eficiencia de los algoritmos existentes. Además, se lleva a cabo una revisión detenida de los algoritmos L^* y Kearns Vazirani, así como de los diferentes oráculos utilizados para la extracción de modelos.

Abstract

This work contributes to the field of Explainable Artificial Intelligence (XAI). Our aim is to enhance algorithms, optimize processes, and conduct a thorough review of the state of the art. Specifically, optimizations are proposed for the extraction algorithms used in the TAYSIR international competition, and the implementation of the Observation Pack algorithm is proposed to enhance the efficiency of existing algorithms. Additionally, a detailed review is conducted on the L^* and Kearns Vazirani algorithms, as well as the various oracles used for model extraction.

Palabras clave

Inteligencia Artificial; Máquina finita determinística; Autómata finito determinístico; Explicabilidad; Oráculo; Inferencia Regular; L*; Árbol de Discriminación

Key words

Artificial Intelligence; Finite state machine; Deterministic finite automata; Explainability; Oracle; Regular Inference; L*; Discrimination Tree

Contents

1 Introduction

Nowadays, information systems increasingly incorporate artificial intelligence agents equipped with the ability to learn, so called learning-enabled components [1]. Learning, in this context, is understood as the system's capability to improve its performance through experience [2]. To achieve this, the agent possesses a representation (model) of its knowledge of the interacting environment, which is "trained" using data and the outcomes of interactions. In this scenario, artificial neural networks (ANN) stand out as one of the most successful models due to their capacity to learn from extensive datasets and their consequent predictive effectiveness [3].

The focus of explainable artificial intelligence (XAI) is to develop artifacts capable of delivering intelligent outcomes along with comprehensible justifications. This means not only optimizing model performance metrics like accuracy but also providing transparent and convincing explanations for the decisions in a human-understandable manner [4].

Despite their efficacy, ANN are often considered opaque models [5], posing challenges in understanding the underlying decision-making processes. This lack of transparency is particularly critical in fields like medicine, risk assessment, and intrusion detection, where human interpretation is paramount [6, 7, 8].

To enhance the explainability of ANN, extensive research has been dedicated to addressing this limitation [9, 10]. One significant approach involves the use of automata, a model that provides a language-independent mathematical foundation for studying dynamical systems. In particular, automata are well-suited for characterizing systems that produce sequences of events corresponding to words in a regular language [11].

In the context of automata-theoretic approaches, the challenge of constructing an automaton when the internal structure of the system is unknown is referred to as identification or regular inference [12]. Practical constraints often make it unfeasible to precisely solve this problem, leading to the exploration of approximate solutions within the Probably Approximately Correct (PAC) framework [13].

In the pursuit of addressing this challenge, the team of Artificial Intelligence and Big Data at Universidad ORT has developed an extraction framework called Neural-Checker [14].

Our goal in this thesis explores the intersection of artificial intelligence and automata theory to advance in the field of verification and analysis of ANNs, contributing both algorithmic advancements and a comprehensive overview of the current state of the art.

Outline

In Chapter 2, we introduce the problem of language identification as a means to provide an overview of automata theory, a crucial concept for this thesis. Chapter 3 presents another families of automata that are analyzed in this work, while Chapter 4 focuses on the algorithmic enhancements made to the Neural-Checker [14] tool. In Chapter 5, we explain in detail tree-based algorithms that have been implemented in the tool. Chapter 6 is dedicated to present various oracles designed for inference algorithms and equivalence of black-box and white-box models. Chapter 7 showcases the results achieved using some of the algorithms in an online competition called TAYSIR [15] in which the team participated in 2023. Lastly, Chapter 8 summarizes the achievements of our work.

2 Learning Regular Languages

2.1 Languages and Automatons

2.1.1 Formal Languages

As seen in [16], a formal language $\mathcal{L}$ is defined as a collection of words (symbol sequences) derived from a non-empty finite set of symbols Σ, referred to as the *alphabet*. It is important to note that not every sequence formed by the alphabet necessarily belongs to the language. For each language, there exists a set of rules that precisely determine which sequences are considered part of it.

Each word formed by an alphabet can be seen as a concatenation of symbols starting from the empty symbol (also seen as the identity symbol) ϵ.

For example, the English language is a set of words that can be formed with symbols of the modern Latin-alphabet. As mentioned before, not all the words that can be constructed by the modern Latin-alphabet belong to the English language. For instance, the sequence “aaaab” can be formed with the symbols from the modern Latin-alphabet but does not belong to the English language.

It is interesting to mention that every alphabet is also a language, more specifically one that only contains unitary symbols.

The empty language, denoted as $\emptyset$, is the language that contains no words at all (not even the empty word, $\epsilon \notin \emptyset$).

The language formed by all the possible sequences over an alphabet Σ is called *universal language* and is denoted by Σ^*.

2.1.2 Regular Languages

Regular languages belong to the same family of formal languages compared to the ones represented by a finite state automaton or regular expressions.

This language family is generated by Type-3 grammars (so called regular grammars), defined in Chomsky’s hierarchy [17].

A language $\mathcal{L}$ over Σ is regular if it can be defined by one of the following rule sets:

Left-regular grammar

- $S \rightarrow \epsilon$
- $S \rightarrow s_1$
- $S \rightarrow s_2 S$

Right-regular grammar

- $S \rightarrow \epsilon$
- $S \rightarrow s_1$
- $S \rightarrow S s_2$

Where S is non terminal and $s_1,\ s_2$ are terminal such that $s_1, s_2 \in \Sigma$.

An example of a regular language S, only containing words ending with a, over the alphabet $\Sigma = \{a,\ b\}$ can be defined with:

- $S \rightarrow a$
- $S \rightarrow aS$
- $S \rightarrow bS$

Note that this language can be represented by the regular expression $(a \mid b)^* a$.

2.1.3 Deterministic Finite Automaton

A deterministic finite automaton (from now on DFA) is a tuple $D = \langle Q,\ \Sigma,\ \tau^*,\ q_0,\ F\rangle$, where:

- Q is a set of states,
- Σ is an alphabet,

- $\tau^* : Q \times \Sigma^* \rightarrow Q$ is the transition function,
- $q_0 \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

Let a sequence $s = s_1 s_2 \ldots s_n$, the definition of $q_n = \tau^*(q_1, s)$ means that $q_i = \tau(q_{i-1}, s_i)$ for $2 \leq i \leq n$. To put it in words, the state q_n is the result of the composition of the function τ for every symbol of the sequence s starting from the state q_1.

Now acceptance can be defined. A DFA $\langle Q,\ \Sigma,\ \tau^*,\ q_0,\ F\rangle$ accepts the sequence s if and only if $q_n = \tau^*(q_0, s)$ and $q_n \in F$.

In the same context, given two states q_1 and q_2, it is stated that q_2 is a reachable state from q_1 if and only if it exists a sequence s so that $\tau^*(q_1, s) = q_2$. A *reachable* state for a DFA is a state $q_i \in Q$ that can be reached from q_0.

Lastly, a DFA is *complete* if and only if the function τ^* is a complete function (τ is defined for every tuple symbol-state). In other words, it is an automaton in which all transitions for each state are defined. Moreover, a DFA is called *partial* when its function τ^* is not a complete function.

The language defined by a DFA D is the set of words that are accepted by D and is called $\mathcal{L}(D)$.

As mentioned before the set of all languages that can be defined by a DFA is the set of all Regular Languages.

2.1.3.1 Example

Consider a scenario where a content management system uses a specific naming convention for articles. The convention requires that each article title begins with the letter 'a' and can be followed by zero or more subsections, each denoted by the letter 'b'. For instance:

- Main article: "a".
- Article with subsections: "abb".

This scenario can be represented with the DFA in Figure 2.1 which represents the language ab^*.

1. $Q = \{q_0,\ q_1\ \ q_2\}$ as the set of states.
2. $\Sigma = \{a,\ b\}$ as the alphabet.
3. τ^* the transition function defined as showed in the 2.1.
4. q_0 as the initial state (indicated by the incoming arrow).
5. $F = \{q_1\}$ as the set of final states (indicated by a double circle).

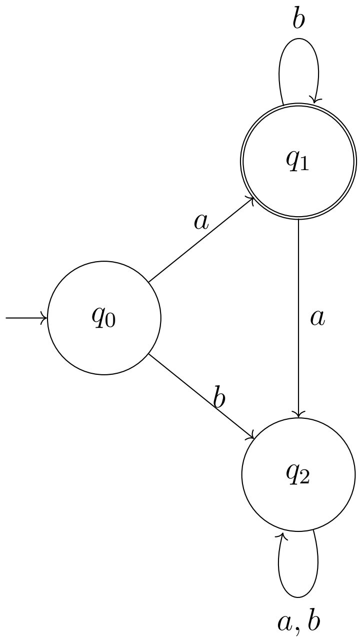


Figure 2.1: DFA which represents the language ab^*

2.1.4 Equivalence

Equivalence in DFAs is defined by: $D_1 \equiv D_2 \iff \mathcal{L}(D_1) = \mathcal{L}(D_2)$ meaning both automatons define the same language. Note that is it possible that two DFAs $D_1 = \langle Q_1,\ \Sigma,\ {\tau^*}_1,\ q_1,\ F_1\rangle$ and $D_2 = \langle Q_2,\ \Sigma,\ {\tau^*}_2,\ q_2,\ F_2\rangle$ so that $D_1 \equiv D_2$ but $Q_1 \neq Q_2\ \vee\ {\tau^*}_1 \neq {\tau^*}_2\ \vee\ q_1 \neq q_2\ \vee\ F_1 \neq F_2$. In other words it is possible for two different DFAs to be equivalent: $\exists\ D_1, D_2\ \textit{so that}\ D_1 \neq\ D_2\ \wedge\ D_1 \equiv D_2$.

Another interesting aspect worth mentioning is that if two DFAs are equal, then they are equivalent, denoted as $D_1 = D_2 \implies D_1 \equiv D_2$. Equivalence in finite automata is fundamental in theoretical computer science. Hopcroft-Karp [18] algorithm is widely known for its effectiveness in determining the equivalence between finite automata.

It is worth noting that there exists a unique minimal automaton that is equivalent to a given one. This concept of minimality is crucial in various aspects of automata theory.

2.1.5 Minimization of a DFA

Minimization/optimization of a DFA involves identifying states whose presence or absence does not affect the language accepted by the automaton. These states can be eliminated or merged without altering the language recognized by the automaton [19]. This process includes:

- **Detection of unreachable states**: states that cannot be reached from the initial state. Eliminating unreachable states has no impact on the language accepted by the automaton.
- **Identification of non-distinguishing states**: states with indistinguishable behavior regarding language acceptance for any input sequence. This concept is formalized by the Nerode congruence, which identifies states that exhibit the same behavior in terms of language recognition. Merging non-distinguishing states reduces the overall number of states without changing the language recognized.
- **Detecting dead states**: states from which no accepting state can be reached. This states contribute no information to the language recognized by the automaton and can be removed unless is required for the DFA to be complete. Moreover, multiple dead states also fit in the category of non-distinguishing states and can be merged following that rule.

There exist algorithms to minimize a DFA, such as Hopcroft's algorithm [20], which is a widely-used method. By reducing the number of states, a minimized DFA processes input more efficiently, leading to faster language recognition.

2.2 Learning regular languages: L*

Grammatical inference is a task where the goal is to learn or infer a grammar (or some device that can generate, recognise or describe strings) for a language of which we are given an indirect presentation through strings, sequences, trees, terms or graphs [12].

There are two settings that the learning processes could adopt, and those are active learning and passive learning [21].

Passive automata learning infers an automaton from a given dataset [22]. Gold demonstrated that the task of deducing a DFA with k states from a provided dataset is proven to be *NP-complete* [23].

In the context of active learning, the learner has the capability to actively choose examples and make membership queries to the teacher. A well known algorithm in the category of active learning is Angluin's L^* [24]. This algorithm is polynomial in the number of states of the minimal DFA and the maximum length of any sequence exhibited by the teacher.

L^* constructs a DFA by interacting with a Minimum Adequate Teacher (MAT) that exposes two operations: a *membership query* (**MQ**), that is a boolean response if a given sequence is accepted by the language known by the teacher, and an *equivalence query* (**EQ**), that is a function that compares the target language and the inferred one, if they are equivalent the test returns true, if not it returns a counterexample (a word belonging to one of the languages but not the other) [21].

The different routines are explained in the following subsections based on [21] and [12].

2.2.1 Initialization and Data Structures

From now on, the target automaton is represented as $\mathcal{M}$ and the inferred DFA as $\mathcal{H}$. The way the algorithm achieves the learning is as follows. It builds a table of observations by interacting with the MAT. This table is used to keep track of which words are and are not accepted by the target language. The construction of this table is done in an iterative way by asking the teacher membership queries of different words in order to fill the Observation Table (OT).

The information that is in the Observation Table has three characteristics:

- a nonempty finite prefix-closed set of strings (every prefix of every member is also a member of the set),
- a nonempty finite suffix-closed set of strings (every suffix of every member is also a member of the set),
- and a finite function that maps a string to either 1 or 0 if it is a member of our target language or not, respectively.

The Observation Table is composed of two sets of rows: the ‘upper’ rows (or top part, that we call **RED** following De la Higuera’s notation [12]), that represent the elements of the prefix-closed set of strings mentioned earlier, and the ‘lower’ rows (or bottom part, that we call **BLUE**), which represent the same elements of this set but concatenated with the set of symbols in the language alphabet. On the other hand, columns represent a suffix-closed set of strings, and each cell represents the membership relationship, both also mentioned earlier. An example of the Observation Table is presented in Table 2.1.

We use two operations of the Observation Table:

- $OT[s]$ represents the **row** in the Observation Table defined by s, where s is a string.
- $OT[s_1][s_2]$ represents the **cell** in the Observation Table defined by the row s_1 and the column s_2, where s_1 and s_2 are strings.

The Observation Table is first initialized by building one **RED** row (for the empty word ϵ) and one **BLUE** row for each symbol in the alphabet Σ (length-one words) as shown in Algorithm 1. Then the iterative process begins.

Algorithm 1 Initialization routine

```
1: procedure L*-INIT($\mathcal{M}$)
2:     $OT \leftarrow$ BUILD-OBSERVATION-TABLE($\mathcal{M}$)
3:     $OT[\epsilon][\epsilon] = \mathbf{MQ}(\epsilon)$
4:     for each $a \in \Sigma$ do
5:         $OT[a][\epsilon] = \mathbf{MQ}(a)$
6:     end for
7: end procedure
```

2.2.2 Properties

In order to make sense out of the table, it needs to comply with two properties. First of all, it needs to be closed. The table is considered closed if, for every row in **BLUE**, there is an equal row in **RED**, as indicated in Algorithm 2.

The second property is consistency. A table is considered consistent if for every pair of rows in **RED** with the same values (same order of 0s and 1s), then all pairs of extensions with the same symbol of the alphabet must have the same row in the table (Algorithm 3). Precisely, a table is consistent if for all v, w in **RED** with $v \neq w$ and for every symbol a in Σ, if $OT[v] = OT[w]$, then $OT[va] = OT[wa]$.

Algorithm 2 Check if OT is closed

```
function Is-Closed(OT)
    for each $v \in$ BLUE do
        if $OT[v] \notin$ RED then
            return $False$
        end if
    end for
    return $True$
end function
```

Algorithm 3 Check if OT is consistent

```
function Is-Consistent($OT$)
    for each $v \in$ RED do
        $equal\_rows \leftarrow$ Get-Rows-With-Equal-Value($v$)
        for each $w \in equal\_rows$ do
            for each $a \in \Sigma$ do
                if $OT[va] \neq OT[wa]$ then
                    return $False$
                end if
            end for
        end for
    end for
    return $True$
end function
```

If the table is not closed, the algorithm moves to **RED** a row in **BLUE** that does not have an equal row in **RED** and adds to **BLUE** all the rows corresponding to the extensions of its associated word with every symbol of the alphabet, as illustrated in Algorithm 4.

Let c be the column for which the inconsistency has been found. To make it consistent, the algorithm expands the original set of suffixes with the symbol that makes their corresponding extensions different (an $a \in \Sigma$ such that $OT[v] = OT[w]$ but $OT[va] \neq OT[wa]$) concatenated with c (ac) as demonstrated in Algorithm 5. This is done to differentiate between the two words that had the same row values.

Algorithm 4 Make OT closed

```
1: procedure CLOSE($OT$)
2:     for each $v \in$ BLUE do
3:         if $OT[v] \notin$ RED then
4:             ADD-TO-RED($v$)
5:             REMOVE-FROM-BLUE($v$)
6:             for each $a \in \Sigma$ do
7:                 ADD-TO-BLUE($va$)
8:                 $OT[va] =$ MQ($va$)
9:             end for
10:        end if
11:    end for
12: end procedure
```

Algorithm 5 Make OT consistent

```
procedure MAKE-CONSISTENT($OT$)
    for each $v \in$ RED do
        $equal\_rows \leftarrow$ GET-ROWS-WITH-EQUAL-VALUE($v$)
        for each $w \in equal\_rows$ do
            for each $a \in \Sigma$ do
                if $OT[va] \neq OT[wa]$ then
                    ADD-OT-COLUMN($ac$)
                    for each $r \in OT$ do
                        $OT[r][a] = \mathbf{MQ}(ra)$
                    end for
                end if
            end for
        end for
    end for
end procedure
```

2.2.3 DFA construction and counterexample processing

Once the table is closed and consistent, the algorithm proceeds to construct the conjectured DFA (Algorithm 6). To build an automaton out of the table, the states are represented by every unique row in **RED**. The final states are those corresponding to the rows w where $OT[w][\epsilon] = 1$, and rejecting states are those rows v where $OT[v][\epsilon] = 0$ (lines 4–7). Finally, the transition function is defined as: $\tau^*(q_v, a) = w$ if $OT[va] = OT[w]$, as showed in lines 8–14.

Algorithm 6 Construct conjectured DFA

```
1: function CONSTRUCT-DFA($OT$)
2:     $\mathcal{H} \leftarrow$ CREATE-DFA($\Sigma$)
3:     for each $r \in$ RED do
4:         if $r \notin \mathcal{H}$ then
5:             $accepts \leftarrow OT[r][\epsilon]$
6:             $s \leftarrow$ CREATE-STATE($\mathcal{H}, s, accepts$)
7:         end if
8:         for each $s \in \mathcal{H}$ do
9:             for each $a \in \Sigma$ do
10:                $\hat{s}\_value \leftarrow OT[sa]$
11:                $\hat{s} \leftarrow$ KEY($\hat{s}\_value$)        ▷ Key denotes the row of $\hat{s}\_value$
12:                CREATE-TRANSITION($\mathcal{H}, a, s, \hat{s}$)
13:            end for
14:        end for
15:    end for
16:    return $\mathcal{H}$
17: end function
```

Once the DFA is built, the algorithm asks the teacher whether it is equivalent to the target one. If the answer is yes, it terminates and returns the learned DFA. If the answer is no, then it receives a counterexample that proves the DFA is wrong, and it proceeds to extend the Observation Table with this new counterexample (Algorithm 7). This extension is done by adding every prefix of the counterexample to **RED**, and for each prefix, its concatenation with every symbol in Σ to **BLUE** (given that the concatenation is not a prefix).

Algorithm 7 Counterexample processing

```
procedure Process-Counterexample($OT, \varphi$)
    $ps \leftarrow$ Get-Prefixes($\varphi$)
    for each $p \in ps$ do
        Add-To-RED($p$)        ▷ Remove from BLUE if it already exists in OT
        Fill-Row($p$)
        for each $a \in \Sigma$ do
            if $pa \notin$ RED then
                Add-To-BLUE($pa$)
                Fill-OT-Row($pa$)
            end if
        end for
    end for
end procedure

procedure Fill-OT-Row($p$)
    for each $c \in columns$ do
        $OT[p][c] =$ MQ($pc$)
    end for
end procedure
```

2.2.4 Putting it all together

Having presented all the necessary functions and procedures included in the L* algorithm, the complete algorithm can now be introduced. We begin by initializing the Observation Table. Subsequently, we generate an hypothesis by executing the closure and consistency procedures on the table. The algorithm then follows with the processing of counterexamples and the generation of a new hypothesis until it is equivalent to the target.

It is noteworthy that, making OT consistent may cause the table to be not closed and making OT closed may cause the table to be not consistent. For this reason, we have to make sure the table is both closed and consistent before making an **EQ**.

Algorithm 8 L* algorithm

```
function L*-ALGORITHM($\mathcal{M}$)
    $OT \leftarrow$ L*-INIT($\mathcal{M}$)
    CLOSE($OT$)
    MAKE-CONSISTENT($OT$)
    $\mathcal{H} \leftarrow$ CONSTRUCT-DFA($OT$)
    $are\_eq, \varphi \leftarrow$ EQ($\mathcal{H}, \mathcal{M}$)
    while $not(are\_eq)$ do
        PROCESS-COUNTEREXAMPLE($OT, \varphi$)
        while $not$ (IS-CLOSED($OT$)) $and\ not$ (IS-CONSISTENT($OT$)) do
            CLOSE($OT$)
            MAKE-CONSISTENT($OT$)
        end while
        $\mathcal{H} \leftarrow$ CONSTRUCT-DFA($OT$)
        $are\_eq, \varphi \leftarrow$ EQ($\mathcal{H}, \mathcal{M}$)
    end while
    return $\mathcal{H}$
end function
```

The time L* consumes depends on the length of the counterexample presented by the teacher [24]. Angluin defines n as the number of states of $\mathcal{M}$ and m as the maximum length of any counterexample presented by the teacher during the running of L*. The total running time can be bounded by a polynomial function of m and n as reported in [24].

2.2.5 Example run

Let's take the DFA of Figure 2.1 as an example to run L*.

First, the algorithm constructs the table as presented in Table 2.1a. As the table is not closed (not every row in **BLUE** has a representation in **RED**), the algorithm proceeds to close it. To do that, the unique element (a) in **BLUE** that has not a representative in **RED** is selected, and moves it to **RED**, adding to **BLUE** its concatenation to every symbol (aa and ab, then the holes are filled). The resulting table can be seen in Table 2.1b.

As the Observation Table is now closed and consistent, an automaton can be built. This automaton is presented in Figure 2.2.

This automaton is then presented to the teacher via the **EQ**, which can be

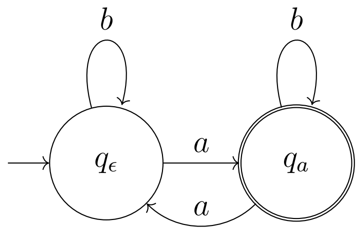


Figure 2.2: DFA in a L*example run.

implemented by the table-filling algorithm [16]. This query results negative, as the regular language that the conjectured automaton represents is not the same as the target one. Let us suppose that the counterexample returned by the teacher is '*ba*'.

Now, the learner proceeds to process the counterexample. This is done by:

- Adding the counterexample and all its prefixes to **RED**. In this case *ba* and *b* are added.
- Adding for each prefix v and for all symbol w, vw to **BLUE**, given that vw is not a prefix of the counterexample. In this case the prefixes are *b* and *ba*; so *ba*, *bb*, *baa* and *bab* are the candidates to be added to **BLUE**. However, since *ba* is the counterexample, only *bb*, *baa* and *bab* need to be added to **BLUE**.

Then holes are filled, resulting in the Table 2.1c.

The table remains closed, however it is not consistent, as two **RED** rows have different resulting rows if they are added a symbol. To be concrete, $OT[\epsilon] = OT[b]$, however $OT[\epsilon a] \neq OT[ba]$. This can be informally interpreted as 'they seem to be the same state in the table, however they are not', so they have to be separated. This separation is achieved by adding the symbol that makes them differ concatenated with the sequence ϵ (the column in which the inconsistency was found) to the columns of the Observation Table (in this case sequence a). The sequence is added, holes are filled, the result is Table 2.1d.

The last table is closed and consistent, the conjectured automaton is finally equivalent to the target one, so **EQ** outputs $\top$, L* finishes and the DFA presented in Figure 2.3, which is equivalent to the target one, is returned.

OT_0	ϵ
ϵ	0
a	1
b	0

(a)

OT_1	ϵ
ϵ	0
a	1
b	0
aa	0
ab	1

(b)

OT_2	ϵ
ϵ	0
a	1
b	0
ba	0
aa	0
ab	1
bb	0
baa	0
bab	0

(c)

OT_3	ϵ	a
ϵ	0	1
a	1	0
b	0	0
ba	0	0
aa	0	0
ab	1	0
bb	0	0
baa	0	0
bab	0	0

(d)

Table 2.1: Observation Tables during a L* run

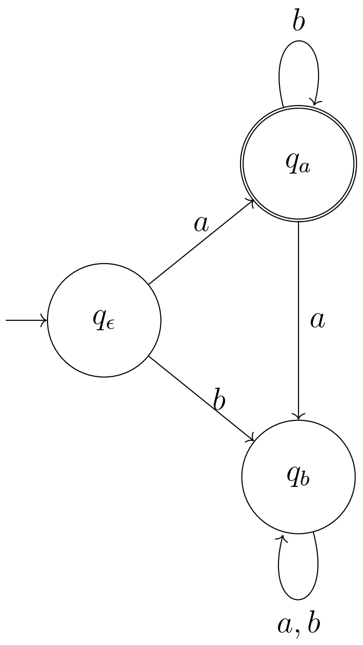


Figure 2.3: Output DFA in a L* example run.

3 Learning Moore & Mealy Machines

3.1 Moore Machines

A Moore Machine, introduced by Edward F. Moore [25], is a type of finite-state machine with inputs and outputs. It can be formally defined as a 6-tuple $\langle Q, \Sigma, O, \tau^*, G, q_0 \rangle$, where:

- Q is a finite set of states,
- Σ is the input alphabet,
- O is the output alphabet,
- $\tau^* : Q \times \Sigma^* \to Q$ represents the transition function
- $G : Q \to O$ is a function that maps each state to an output symbol,
- $q_0 \in Q$ is the initial state.

In Moore Machines, like other finite-state machines, the output always depends on the current state, but not on the current input. These machines are typically characterized by being both *deterministic* and *complete*, meaning that for any given state and input, the next state is defined and unique, and the output is also uniquely determined.

An example of a Moore Machine is presented in Figure 3.1. This automaton models a simple elevator control, with 3 floors: Ground (GR), First Floor (F1) and Second Floor (F2). Transitions between states are triggered by symbolic inputs, where 'Up' represents a request to move up, 'Down' represents a request to move down, and 'Emergency' indicates an emergency run to GR, and once there the elevator is blocked. This Moore Machine can be described by:

1. Q = $\{q_0, q_1, q_2, q_3\}$
2. $\Sigma = \{$ *"Up", "Down", "Emergency"*$\}$ and ϵ being the empty sequence

3. $O = \{GR,\ F1,\ F2\}$
4. τ^* as represented graphically in Table 3.1
5. $G = \{q_0 \to GR,\ q_1 \to F1,\ q_2 \to F2,\ q_3 \to GR\}$
6. $q_0 \in Q$ is the initial state.

Moore Machines can exhibit two different behaviours. First, they can be regarded as transducers, producing a sequence of output symbols ρ_{out} given a sequence of input symbols ρ_{in}. Alternatively, one may seek only the *last symbol* produced by the machine when provided with a sequence of input symbols. The latter is the one we will focus on.

A DFA can be seen as a special case of a Moore Machine where we observe the last symbol and its output alphabet is binary, say $O = \{\top, \bot\}$. In this context, the last symbol returned by the machine can be interpreted as acceptance ($\top$) or rejection ($\bot$).

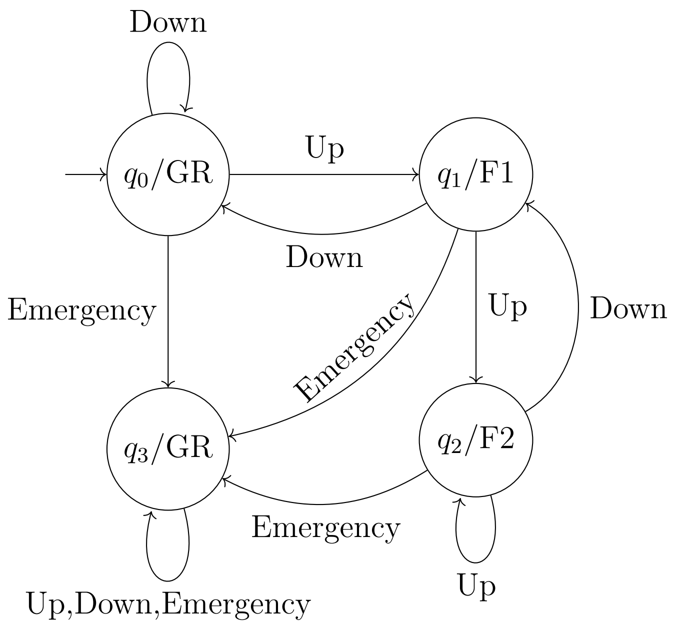


Figure 3.1: Example Moore Machine.

ϵ	Up	Down	Emergency
q_0	q_1	q_0	q_3
q_1	q_2	q_0	q_3
q_2	q_2	q_1	q_3
q_3	q_3	q_3	q_3

Table 3.1: Table of transition function τ^* of automaton in Figure 3.1

3.1.1 Extending L* for Moore Machines

The L* algorithm enables the inference of a DFA from a black-box target by interacting with the MAT teacher. The MAT essentially informs us whether the target accepts or rejects specific input sequences. To extend this concept to work with Moore Machines, one approach is to consider the comparison of the *last symbol* output and evaluate it against the hypothesis model.

It is evident that when considering a Moore Machine with a binary output alphabet, such as $O = \{\top, \bot\}$, the process of comparing the last symbols is fundamentally analogous to what the MAT accomplishes in DFA L*. In other words, this extension retains compatibility with DFAs.

Following this, we further explore these modifications, with a particular focus on the tables, MATs, and adaptations to the learner algorithm for this extended L* approach customized for Moore Machines.

3.1.1.1 A Moore Machines L* run

Consider the Moore Machine defined in Figure 3.1 and its transition function τ^* specified in Table 3.1. Additionally, refer to the L* algorithm outlined in section 2.2, which serves as the foundation for this process.

Firstly, the algorithm constructs the table presented in Table 3.2a. Here, it can be observed that the initial distinction between DFA L* and Moore Machines L*: the values in the table do not represent $\{\top, \bot\}$ (rejection and acceptance) but rather the symbols in the output alphabet of the Moore Machine, which in this case are $\{GR, F1, F2\}$.

Next, following the algorithm, we check if the table is closed, which it is not.

Consequently, the algorithm proceeds to close it, and the resulting table can be seen in Table 3.2b. However, even after closing, the table remains not closed. Therefore, the algorithm once more attempts to close the table, resulting in Table 3.2c.

After the table is closed the algorithm checks if the table is consistent. If not it proceeds to make it consistent in the same way as the DFA L*.

As the Observation Table is now closed and consistent, an hypothesis automaton can be built. This automaton is presented in Figure 3.2

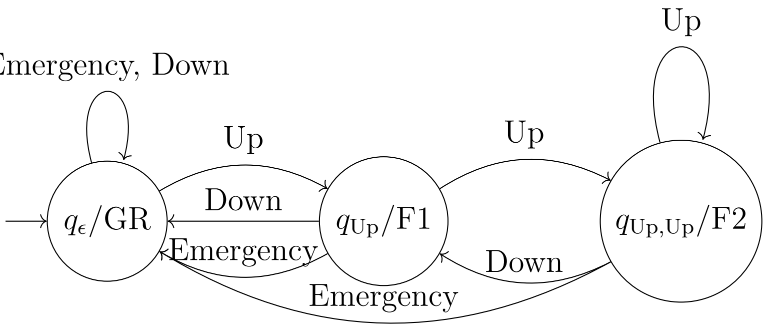


Figure 3.2: Hypothesis Moore Machine

This automaton is then presented to the teacher via the **EQ**. The teacher returns a counterexample, showing that the hypothesis is not the same as the target. Let us suppose that the counterexample returned by the teacher is 'Emergency,Up'[1]. We can see that the output of the target given the sequence 'Emergency,Up' is GR, but the output of the hypothesis is $F1$

Now, the learned proceeds to process the counterexample, resulting in Table 3.2d

The table remains closed, however is not consistent, since $OT[Emergency, Up] = OT[\epsilon]$ but $OT[Emergency, Up, Up] \neq OT[\epsilon, Up]$. This separation is achieved by adding the symbol that makes them differ to the columns of the Observation Table. The result is Table 3.2e

The last table is both closed and consistent, an hypothesis automaton is built and presented to the teacher. Consequently, **EQ** outputs $\top$, indicating that the conjectured automaton is finally equivalent to the target one, so the L* algorithm concludes. The Moore Machine showed in Figure 3.3, which is equivalent to the target machine, is then returned.

[1]For ease of reading, concatenated symbols are coma separated, so 'Emergency,Up' represents a sequence of the concatenated symbols 'Emergency' and 'Up'.

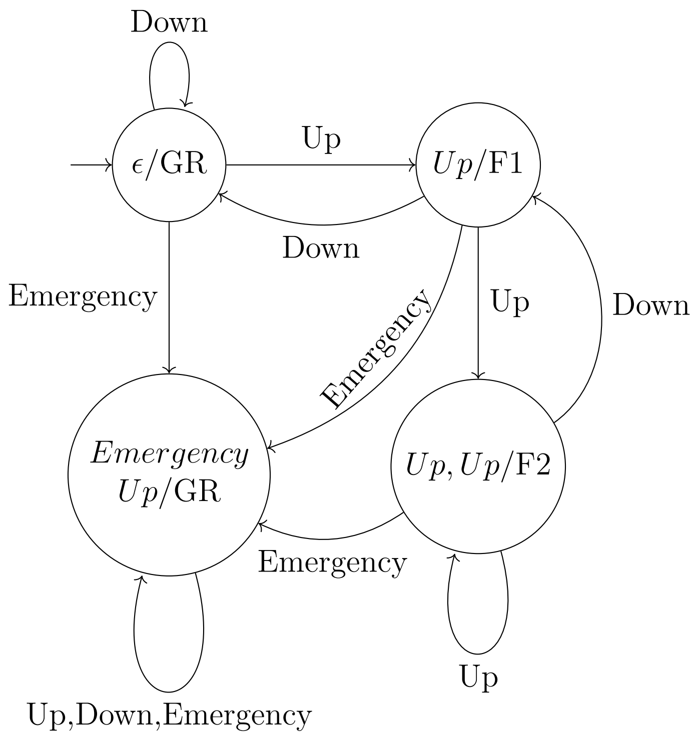


Figure 3.3: Moore Machine returned by L*

OT_0	ϵ
ϵ	GR
Up	$F1$
$Down$	GR
$Emergency$	GR

(a)

OT_1	ϵ
ϵ	GR
Up	$F1$
Up, Up	$F2$
$Up, Down$	GR
$Up, Emergency$	GR
$Down$	GR
$Emergency$	GR

(b)

OT_2	ϵ
ϵ	GR
Up	$F1$
Up, Up	$F2$
$Up, Down$	GR
$Up, Emergency$	GR
Up, Up, Up	$F2$
$Up, Up, Down$	$F1$
$Up, Up, Emergency$	GR
$Down$	GR
$Emergency$	GR

(c)

OT_3	ϵ
ϵ	GR
Up	$F1$
Up, Up	$F2$
$Emergency, Up$	GR
$Up, Down$	GR
$Up, Emergency$	GR
Up, Up, Up	$F2$
$Up, Up, Down$	$F1$
$Up, Up, Emergency$	GR
$Emergency, Up, Up$	GR
$Emergency, Up, Down$	GR
$Emergency, Up, Emergency$	GR
$Down$	GR
$Emergency$	GR

(d)

OT_4	ϵ	Up
ϵ	GR	$F1$
Up	$F1$	$F2$
Up, Up	$F2$	$F2$
$Emergency, Up$	GR	GR
$Up, Down$	GR	$F1$
$Up, Emergency$	GR	GR
Up, Up, Up	$F2$	$F2$
$Up, Up, Down$	$F1$	$F2$
$Up, Up, Emergency$	GR	GR
$Emergency, Up, Up$	GR	GR
$Emergency, Up, Down$	GR	GR
$Emergency, Up, Emergency$	GR	GR
$Down$	GR	$F1$
$Emergency$	GR	GR

(e)

Table 3.2: Observation Tables during a Moore Machines L* run

3.2 Mealy Machines

A Mealy Machine is a type of finite-state machine named after George H. Mealy, who introduced the concept in the following article [26].

In contrast to Moore Machines, Mealy Machines determine their output values based on both their current state and the current input, rather than relying solely on their current state.

Mealy Machines can be formally defined as a 6-tuple $\langle Q, \Sigma, O, \tau^*, G, q_0 \rangle$, which includes the following components:

- A finite set of states Q,
- an input alphabet Σ,
- an output alphabet O,
- $\tau^* : Q \times \Sigma^* \rightarrow Q$, representing the transition function between states,
- $G : Q \times \Sigma \rightarrow O$, denoting the output function that maps pairs of (state, input symbol) to the corresponding output symbol.
- $q_0 \in Q$ is the initial state.

It is worth noting that the transition function τ^* and the output function G can be unified into a single function $T : Q \times \Sigma^* \rightarrow Q \times O$.

Similar to Moore Machines, Mealy Machines can exhibit two distinct behaviors: they can return all the output symbols for every transition the machine undergoes, given the input, or they can return only the last output symbol. As Moore and Mealy Machines are both types of finite state machines, they are equally expressive and are capable of parsing regular languages [**?**].

An example of a Mealy Machine is presented in Figure 3.4. This automaton models the same elevator system as the Moore Machine presented in Figure 3.1. It can be formally defined by:

1. Q = $\{q_0, q_1, q_2, q_3\}$
2. Σ = {*"Up"*, *"Down"*, *"Emergency"*} and ϵ being the empty sequence
3. O = {*GR, F1, F2*},

4. τ^* as represented graphically in Table 3.3a
5. G as represented in Table 3.3b
6. $q_0 \in Q$ is the initial state.

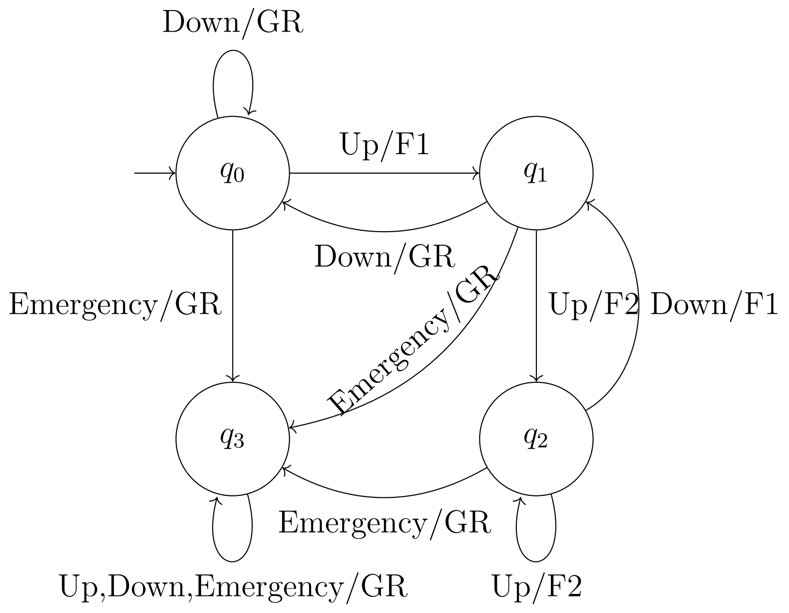


Figure 3.4: Example Mealy Machine.

ϵ	Up	Down	Emergency
q_0	q_1	q_0	q_3
q_1	q_2	q_0	q_3
q_2	q_2	q_1	q_3
q_3	q_3	q_3	q_3

(a) Table of transition function τ^*

ϵ	Up	Down	Emergency
q_0	F1	GR	GR
q_1	F2	GR	GR
q_2	F2	F1	GR
q_3	GR	GR	GR

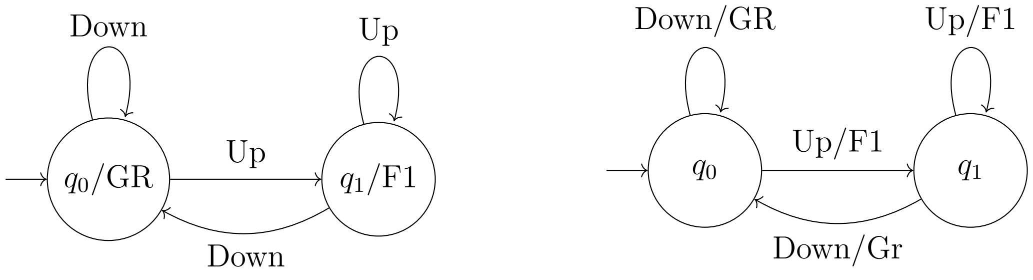
(b) Table of output function map G

Table 3.3: Table of functions of automaton in Figure 3.4

3.2.1 Learning Mealy Machines via Moore Machines

It is possible to convert a Mealy Machine to a Moore Machine and vice versa, in a fairly simple process [27, 28]. If we compare Figure 3.1 and Figure 3.4 it can be seen that they are quite similar. From this observation, we can create a simple algorithm for converting a Moore Machine to a Mealy Machine by adding the output symbol of the state it goes to, to every transition. For example, in state q_0 we have a transition for Up that goes to state q_1 which outputs the symbol $F1$. To convert it to a Mealy Machine, we can take the output symbol $F1$ from state q_1 and add it to the transition going from q_0 to q_1 through the symbol Up.

This process generates a Mealy Machine, but does not ensure that the resulting Mealy Machine is minimal. Mealy Machines can represent the same language using less states than a Moore Machine as they can use inputs to determine outputs without needing a state change. However, this may not always be the case, depending on the input-output relation and the state encoding.



(a) Simplified elevator control Moore Machine

(b) Simplified elevator control converted to Mealy Machine

Figure 3.5: Illustration of conversion of Moore to Mealy and minimal Mealy machines

An example of this can be observed in Figure 3.5a where we present a simplified version of the elevator control shown in Figure 3.1. In this simplified version we have only two floors: ground (GR) and first floor (F1) with the options to go either *Up* or *Down*. When applying our algorithm to convert this Moore Machine into a Mealy Machine, the resulting diagram is illustrated in Figure 3.5b. Nevertheless, it is important to note that this Mealy Machine is not minimal. The minimal Mealy Machine that represents the same language can be seen in Figure 3.6.

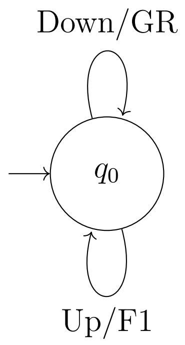


Figure 3.6: Minimal Mealy Machine of the simplified elevator control

3.2.1.1 Moore to Mealy Algorithm

The algorithm to convert a Moore Machine into a Mealy Machine is now presented.

3.2.1.2 Initialize table for Mealy Machine

Algorithm 9 initializes the Mealy Machine table (MT) and the state mapping (`state_map`). The Mealy Machine table represents transitions in the Mealy Machine, while the state mapping associates Moore Machine states with their corresponding Mealy Machine states. $\mathcal{M}$ represents the Moore Machine we want to convert.

To properly construct the Mealy Machine table and ensure the resulting Mealy Machine is minimal, we adhere to the following property, which establishes a relationship between Moore States and Mealy States: $\forall q, q' \in \mathcal{M}, \forall s \in \Sigma, \tau(q, s) = \tau(q', s) \Rightarrow q \wedge q'$ can be represented by the same Mealy State.

The algorithm traverses all states in the Moore Machine to build the table. For each state, it creates a `transition_info` structure, which includes, for every symbol in the alphabet, a set containing: the symbol, the name of the state in the Moore Machine it transitions to (referred to as `next_state`), and the output symbol of that transition state.

The `transition_info` will be the key in the MT, so, if two states have the same `transition_info` then they are the same and we do not add it. The value in the table will be the name of the new Mealy State. In the algorithm presented, we use the same name as the Moore State we are processing, but it can be any name.

The purpose of the `state_map` is to map the Moore States names into the new Mealy States names. This is necessary because the `next_state` stored in the `transition_info` is a Moore State. Therefore, when adding transitions later, we must follow the `transition_info`. However, we need to map the name in the `transition_info` to the corresponding Mealy State.

Adding a new state in the `state_map` goes as follow: if the `transition_info` was not in the MT, we will map the Moore State to the same name we choose for the Mealy State (in this case they have the same name). If the `transition_info` is already in the MT, then the `state_map` will save, for the Moore State we are processing, the Mealy State that has the same `transition_info` indicating that they represent the same state in the Mealy Machine.

Algorithm 9 Initialize table for Mealy Machine

```
1: function INIT-MT($\mathcal{M}$)
2:     $MT \leftarrow$ BUILD-MEALY-TABLE()
3:     $state\_map \leftarrow$ BUILD-STATE-MAP()
4:     for each $state \in \mathcal{M}$ do
5:         $transition\_info \leftarrow$ CREATE-TRANSITION-INFO
6:         for each $symbol \in \mathcal{M}.alphabet$ do
7:             $next\_state \leftarrow state.next\_state\_for(symbol)$
8:             $transition\_info.append((symbol, next\_state.value, next\_state.name))$
9:         end for
10:        if $transition\_info \notin MT$ then
11:            $MT[transition\_info] \leftarrow state.name$
12:            $state\_map[state.name] \leftarrow state.name$
13:        else
14:            $state\_map[state.name] \leftarrow MT[transition\_info]$
15:        end if
16:    end for
17:    return $MT, state\_map$
18: end function
```

3.2.1.3 Build Mealy Machine States

Now we have to implement a function that creates a list of Mealy states (Algorithm 10). For each state name in the MT, a Mealy state is constructed using the `Create-Mealy-State` function.

Algorithm 10 Create Mealy States

```
1: function BUILD-MEALY-STATES($MT$)
2:     $states\_list \leftarrow$ CREATE-STATES-LIST()
3:     for each $state\_name \in MT.values()$ do
4:         $states\_list[state\_name] \leftarrow$ CREATE-MEALY-STATE($state\_name$)
5:     end for
6:     return $states\_list$
7: end function
```

3.2.1.4 Add Transitions to Mealy States

An algorithm now is needed in order to add transitions to the Mealy states based on the information stored in the Mealy Table (Algorithm 11), state mapping (`state_map`), and the list of Mealy states (`states_list`).

Algorithm 11 Add Transitions to the states

```
1: function ADD-TRANSITIONS($MT, state\_map, states\_list$)
2:     for each $state\_tranistions, state\_name \in MT$ do
3:         $current\_state \leftarrow states\_list[state\_name]$
4:         for each $symbol, output, next\_moore\_state \in state\_transitions$ do
5:             $next\_mealy\_state \leftarrow states\_list[state\_map[next\_moore\_state]]$
6:             CREATE-TRANSITION($current\_state, symbol, output, next\_mealy\_state$)
7:         end for
8:     end for
9: end function
```

3.2.1.5 Putting it all together

The final algorithm, `Convert-Moore-To-Mealy`, orchestrates the entire conversion process. It uses the previously initialized Mealy Table (Algorithm 9), created Mealy states (Algorithm 10), and added transitions (Algorithm 11) to construct the Mealy Machine.

Algorithm 12 Convert Moore to Mealy machine

```
function CONVERT-MOORE-TO-MEALY($\mathcal{M}$)
    $MT$, state_map ← INIT-MT($\mathcal{M}$)
    $states\_list$ ← BUILD-MEALY-STATES()
    ADD-TRANSITIONS($MT$, state_map, $states\_list$)
    $mealy\_machine$ ← CONSTRUCT-MEALY($states\_list$)
    return $mealy\_machine$
end function
```

4 Algorithmic improvements

4.1 Neural-Checker

The three authors of this thesis actively contribute to Neural-Checker [14], a tool developed by the Artificial Intelligence and Big Data team of Universidad ORT. Its main goal is to provide implementations for the structures needed for working in the Model Extraction Framework and enable the explainability and checking of complex systems in a black box context.

The codebase is separated into two principal libraries: pythautomata and pyModelExtractor as we can see in Figure 4.1. The first one contains the different automata definitions and the second one contains the different learning algorithms. There are also other private repositories for experiments and benchmarks.

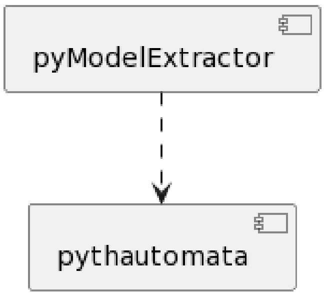


Figure 4.1: Component diagram

From the beginning, we were interested in finding optimization opportunities in the existant automata learning algorithms of the tool and also contribute with other learning algorithms. The execution of extraction algorithms was conducted on specific hardware and software configurations.

Model extractions were conducted on a JupyterHub workstation provided by Universidad ORT with the following hardware configuration:

- **Processor:** Intel(R) Xeon(R) W-2195 CPU @ 2.30GHz (with a boost clock of 4.30GHz).
- **RAM:** 503 gigabytes.

- **Storage:** 1.8 terabytes (overlay).

The software environment used for model extractions is detailed below:

- **Operating System:** Ubuntu 18.04.6 LTS.
- **Development Environment:** Python 3.9.7.
- pythautomata version: 0.38.5.
- pyModelExtractor version: 0.36.6.

All the contributions made can be found in a GitHub repository[1]. This repository contains a user guide documenting all the contributions made to the tool. Future work will involve adding the remaining features implemented by other authors.

To initiate our interaction with the tool, we started by implementing Moore Machines and Mealy Machines. This explains our discussion about them in the last chapter, wherein we introduced new definitions and, of course, adapted extraction algorithms to be compatible with these finite state machines.

Following that, we dived into our initial algorithm: L*. Our aim was to extend its applicability to other finite state machines and to improve the efficiency of this widely used extraction algorithm. The subsequent section elaborates on these optimizations.

4.2 L*

In this section, optimizations and improvements of the L* algorithm are presented, all included in the Neural-Checker tool.

4.2.1 Optimizations

Before our contributions to the tool, the algorithm was implemented similar to what was presented in the algorithm overview. Upon analysis, we recognized that key procedures, namely `Close` and `Make-Consistent`, could benefit from optimization.

[1] `https://github.com/neuralchecker/Neural-Checker-User-Guide`

In the first place, the `Close` operation takes $\mathcal{O}(n \times m)$ in the best, average, and worst cases, where n is the size of **BLUE** and m is the size of **RED**. This could be improved by maintaining a set of all the values of **RED**, implemented with a hash table. This allows us to achieve an average-case time complexity of $\mathcal{O}(n + m)$, which drastically improves performance.

Secondly, we analyze the `Make-Consistent` procedure. To identify inconsistencies, the procedure iterates through all the rows in **RED** and invokes the `Get-Rows-With-Equal-Value` function. This function essentially involves, for each **RED** entry, another iteration through **RED** to identify rows with the same value as the current row. Furthermore, it traverses these identified rows to check if, when concatenated with some symbol from Σ, they do not have the same value in OT. This process has a significant computational cost, and there is an alternative approach to address this issue.

To solve it, a hash table can be used to store the different values as keys and the corresponding row of **RED** as elements. Only the first unique row is added to the hash table, the repeated tuple (key, value) is ignored. This is because we know that if two values with the same row are consistent, then they represent the same state, so it is not necessary to compare a third with both of them because it is redundant.

When traversing **RED**, if the value of the current row is not present in the hash table, then it is stored with the value being the current row. If the value is already present, then it needs to be checked if the element stored in that key is consistent with the current row. If there is an inconsistency, it is returned; otherwise, the traversal of **RED** continues.

Consider the following example: three rows in **RED** a, b, and ba, all of which have the same value. Thus, the inconsistency has to be checked between all of them. Without this optimization, what would have been done is to compare each of them with the rest, resulting in $\mathcal{O}(m^2)$ complexity in the worst case. However, using the hash table, if a is encountered while traversing **RED**, it will be added to the hash table. Then, b is discovered and compared to the row stored in **RED** (which represents a). If they are consistent, the process can continue to find ba since it is not necessary to add b to the hash table because it represents the same state as a. In this case, the comparison only needs to be done with the row in the hash table (a). This makes the entire operation of finding inconsistencies be done in $\mathcal{O}(m)$ worst case.

4.2.2 General L*

Another improvement we made to enhance L* was to generalize the algorithm for other types of finite state machines. Before, separate algorithms were implemented exclusively for DFA and others for different types of state machines.

Recognizing this, we considered it advantageous to develop a unified algorithm. Consequently, we initiated the analysis to extend the algorithm to incorporate both Moore Machines and Mealy Machines, as detailed in Chapter 3.

4.2.3 Partial approach

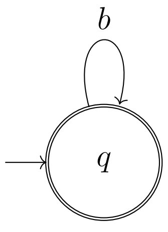
As mentioned earlier, the L* algorithm consistently maintains the Observation Table as a prefix, ensuring closure and consistency. However, when the algorithm is stopped by time bounds, various approaches can be considered, as mentioned in [29].

One option is to return the last built model although it failed an **EQ**. It is crucial to note that this model is distinctly different from the target due to the presence of a counterexample. Furthermore, adopting this approach guarantees that the resulting model is complete. However, it presents a problem: if the run never encountered an **EQ**, no model will be returned.

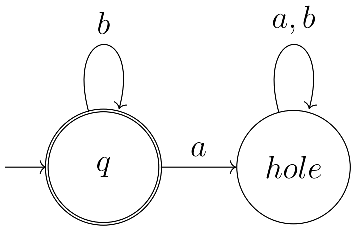
Moreover, this approach does pose a challenge. What if the execution time is predominantly consumed in the closure and/or consistency operations?

This would make the algorithm throw away all the progress since the last **EQ** was made. To address this challenge, the concept of a *partial model* is introduced. A *partial automaton* is an automaton that may not have all transitions defined. When a non-defined transition is encountered, the automaton simply rejects the sequence, utilizing what we refer to as a **hole** state. A **hole** state is a state used for all transitions that are not explicitly defined, where all transitions from this state lead back to itself. This **hole** state is a *non-acceptance* state. Figure 4.2a illustrates a DFA in which the state q does not have defined the a transition. Nonetheless, internally, we can handle this DFA as a complete DFA by employing the **hole** state, as can be seen in Figure 4.2b.

At this point, if the algorithm reaches the time bound, it generates an automaton based on the current state of the table rather than returning the last constructed model.



(a) Partial DFA example.



(b) Complete DFA example using the **hole** state.

Figure 4.2: Illustration of the **hole** state in a real DFA example.

This approach may lead to the construction of an automaton from a table that might not be closed or consistent. To tackle this issue, we can apply the recently introduced concept of a *partial automaton*.

When the table is not closed it could imply the existence of values in **BLUE** that may not have a corresponding state in **RED**. In other words, there is a transition that does not have a target state. To solve this, the partial automaton defines such transitions to the **hole** state.

When the table is no consistent, the partial translation will ignore this inconsistency and assume all the inconsistent sequences in **RED** translate to an equivalent state.

This approach is an improvement based on the assumption that each modification to the table will guide us towards a more precise model compared to the target. It allows us to work with large models such as Recurrent Neural Networks, where the counterexamples may be long. Without imposing a time limit on the generation process, ensuring closure and consistency in the Observation Table could potentially consume a significant amount of time.

It is important to note that this enhancement does not alter the L* algorithm; rather, it modifies the Observation Table Translator responsible for constructing a target model from an Observation Table. This algorithm's applicability extends to any model that can be completed from an Observation Table, including Moore or Mealy Machines.

4.2.4 Restart approach

The introduction of the partial approach has led to another idea to help us learn over large target models. The *restart* approach allows to initiate a L* run with a pre-filled Observation Table. This approach helps when learning over large models as it enables us to save the last Observation Table obtained in a previous run (with the same target model) and continue the run without having to start with an empty Observation Table. Thus, we now can start a run without having to learn all the sequences from previous iterations. This allows us to resume runs and derive multiple *partial* models, enabling us to measure the accuracy of each and retain the best-performing one.

Note that the Observation Table can potentially be not closed or not consistent. This does not pose a problem, since the algorithm will just continue closing the table and making it consistent.

Additionally, the *restart* technique plays a key role in refining the accuracy of learnt models, especially when employing non-deterministic **EQ** strategies like Random Walk or PAC[2]. This method facilitates resuming a run with parameter adjustments, providing a practical way to optimize and enhance the precision of the acquired models.

It is crucial to note that using the *restart* without the *partial model* in time bounded L^* may result in duplicate models, as the Observation Table may not differ between runs, and there may have been no **EQ** in one run.

4.3 Benchmarks

To assess the effectiveness of these improvements, we employed Nicaud's Automaton Generator [30] that generates automata of a given *nominal size* for benchmarking purposes, which it was already implemented in the Neural-Checker tool.

4.3.1 L^* optimization

In order to measure and analyze the efficiency of the proposed versions of L^* algorithms compared to the previous implementation, three experiments were conducted to determine if the optimizations are noticeable.

[2]These strategies are employed in active black-box model extraction algorithms, where **EQ** cannot be guaranteed with 100% security. Chapter 6 provides a detailed discussion of each oracle for both black-box and white-box models.

4.3.1.1 Experiment 1

In this experiment, DFAs and Moore Machines of 200, 500 and 1000 states were generated using a binary alphabet. The algorithms tested were the previous L* version (DFA_L*), L* for Moore Machines (MooreMachines_L*), and the new General L* set to learn DFAs (General_DFA_L*) and Moore Machines (General_MooreMachines_L*).

As it can be observed in Figure 4.3, the new version of the L* algorithm for DFAs outperforms the older version, and this difference increments as the number of states of the target increases. However, in the case of Moore Machines, the difference is not noticeable, as both algorithms already have the optimizations mentioned before. Therefore, it can be concluded that the optimizations and enhancements implemented in the algorithms result in significantly faster execution times.

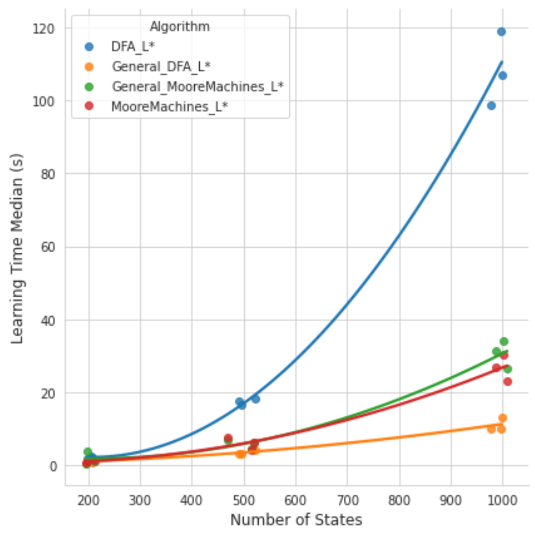


Figure 4.3: Experiment 1 execution time.

4.3.1.2 Experiment 2

For experiment two, we aimed to analyze the behavior of the new, more efficient General L* algorithm across a higher number of states in the target, for learning both DFAs (General_DFA_L*) and Moore Machines (General_MooreMachines_L*). To achieve this, DFAs and Moore Machines with 2000, 3000, and 5000 states were generated.

As showed in Figure 4.4, the time curves for both algorithms continue to increase. However, a significant difference in learning time between DFAs and Moore Machines becomes apparent. We suspect that this discrepancy comes from the implementation details: Moore Machines, as implemented in the tool, involve more abstraction layers, which as discussed in section 5.5 can lead to longer execution times.

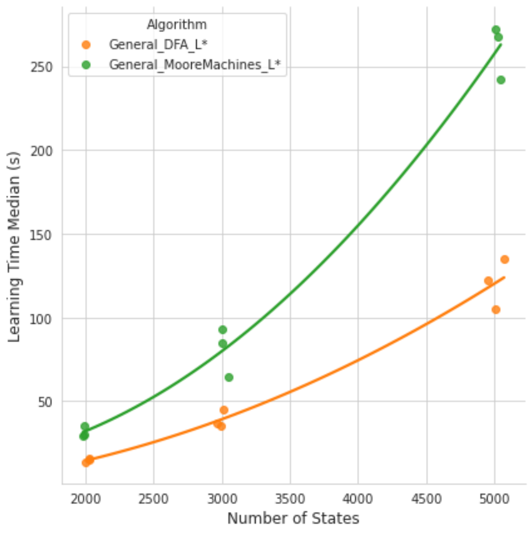


Figure 4.4: Experiment 2 execution time.

4.3.1.3 Experiment 3

In this experiment, we seek to evaluate the impact of alphabet size on efficiency of all algorithms. A DFA and a Moore Machine of 100 states were generated using alphabets of sizes 2, 16, 32, 64 and 128.

The Moore Machines algorithms maintained consistent performance. However, unexpectedly, as the alphabet size increased, the General DFA L* algorithm showed a tendency to require more time and the results are similar to the original DFA L*.

It is imperative to conduct further investigation on future works into the underlying causes of this behavior.

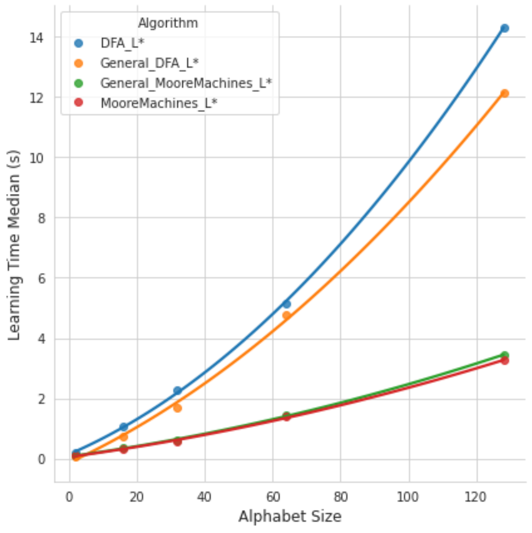


Figure 4.5: Experiment 3 execution time.

4.3.2 Restart & Partial approach

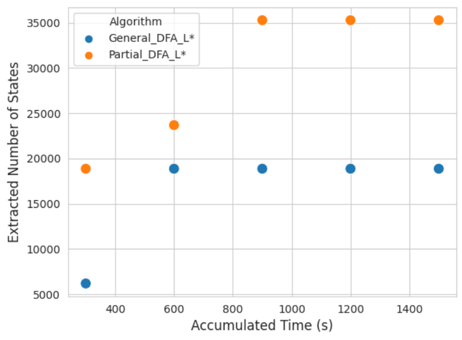
For this experiment, we set two instances of the General DFA L* algorithm with the objective of extracting a 50.000 states DFA target. One of these instances utilized the *partial* approach (for ease of reading we call it partial L*), as outlined in subsection 4.2.3. Both algorithms had a time bound of five minutes (300 seconds), and returned the last DFA they constructed.

To continue with the run, we implemented a *restart* approach by using the last OT obtained. We repeated this process five times. To show the experiment, we track the Accumulated Time of each algorithm, so for example, after the first restart finished (second run) the Accumulated Time of an algorithm is 600 seconds.

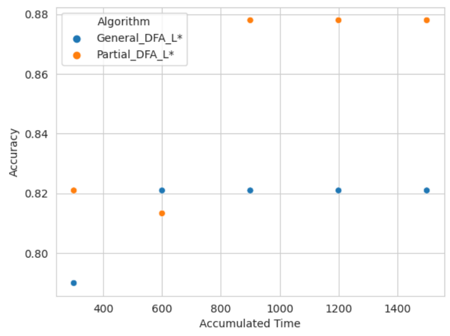
Analyzing the results, in Figure 4.6a, we can observe that the partial approach consistently delivers a higher number of extracted states compared to the General L* in each restart iteration. It can be noted that after 900 seconds of Accumulated Time, the partial approach stops growing at approximately 35.000 states. A possible explanation for this behaviour is that the algorithm is blocked at building a new hypothesis to make an **EQ**, a procedure that takes a lot of time when the number of states is high, thus making no changes to the OT. This can also be the explanation for the General L* not achieving to extract any more states. This should be further explored in future works.

In Figure 4.6b it can be observed that at 600 seconds of Accumulated Time, the partial model has a slightly lower accuracy than the complete one, although the first has more states. This scenario is plausible to happen, but not common. Furthermore, it can be seen that the model learned from the partial L^* achieves an accuracy of over 0.82 in the first iteration, whereas the General L^* achieves it in the second iteration. Moreover, after 900 seconds, the model learned by the partial L^* achieves 0.88 accuracy, a feat not accomplished by the General L^* in this experiment. The results obtained in this experiment validates that models extracted from partial L^* exhibit higher accuracy.

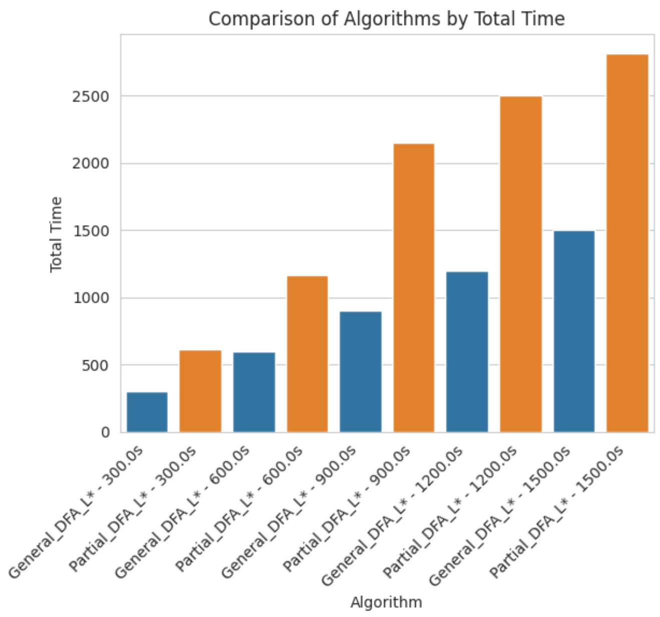
As mentioned earlier, both algorithms are stopped by time bounds. Partial L^*, in contrast to General L^*, after it is stopped by time it proceeds to build the model resulting in additional processing time as it can be seen in Figure 4.6c.



(a) Restart & Partial experiment extracted states by accumulated run time



(b) Restart & Partial accuracy of extracted models by accumulated run time



(c) Restart & Partial total time of execution per algorithm by accumulated time

Figure 4.6: Experiment 5 results.

5 Implementing Discrimination Tree-Based Learning Algorithms

Up to this point, we have discussed one of the two predominant data structures employed in an active automata learning context: the Observation Table. The second one, discrimination trees, is presented in the following chapter.

5.1 Learning with a Discrimination tree

Kearns and Vazirani [31] proposed employing a decision tree, referred to as a discrimination tree, for classification in the context of active automata learning. This approach is crucial for achieving efficiency in the learning process due to the inherent redundancy-freeness of discrimination trees [32].

To formalize, let $\mathcal{M}$ represent the target automaton, and $size(\mathcal{M})$ denote the number of states in $\mathcal{M}$. The algorithm's core concept involves the continuous exploration of new states within $\mathcal{M}$, specifically those states that exhibit distinct behavior from the ones already discovered. The algorithm operates in phases, wherein each phase involves constructing a tentative hypothesis automaton $\mathcal{H}$ with states corresponding to the presently discovered states of $\mathcal{M}$.

Kearns and Vazirani's algorithm is able to learn a model by using membership queries and equivalence queries. In each phase of the algorithm an **EQ** is made on $\mathcal{H}$. The counterexample from this **EQ** allows the algorithm to use **MQ** to discover a new state of $\mathcal{M}$. The algorithm finishes when $\mathcal{H} \equiv \mathcal{M}$.

5.1.1 Formal Notation

Let Σ be an input alphabet. A discrimination tree is a directed binary tree $\mathcal{T}$, where:

- The set of nodes is denoted by $\mathcal{N}_{\mathcal{T}}$, and can be written as the disjoint union of the set of inner nodes $\mathcal{D}_{\mathcal{T}}$ and the set of leaves $\mathcal{S}_{\mathcal{T}}$.
- The designated root node is denoted by $r_{\mathcal{T}} \in \mathcal{N}_{\mathcal{T}}$.

- Each inner node n is labeled with a discriminator v, referred to via $n.discriminator$.
- Each inner node has exactly two children, a 0-child and a 1-child. For $n \in \mathcal{D}_{\mathcal{T}}$, the 0-child is referred to via $n.children[0]$.

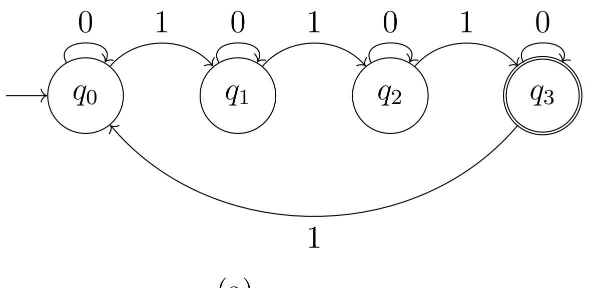
5.1.2 Access Strings and Distinguishing Strings

The learning algorithm mantains a set $\mathcal{S}$ consisting of at most $size(\mathcal{M})$ **state access strings**, and a set $\mathcal{D}$ of **distinguishing strings** in order to discover information about the states of $\mathcal{M}$ [31].

An access string is a string $s \in \mathcal{S}$ so that when it is executed from the start state of $\mathcal{M}$ leads to a unique state. We denote $\mathcal{M}[w], \forall w \in \Sigma^*$ to the operation of executing any string from the start state of $\mathcal{M}$. For each pair of strings $s, s' \in \mathcal{S}$ such that $s \neq s'$ there is a distinguishing string $d \in \mathcal{D}$ such that only one of sd and $s'd$ reaches an accepting state of $\mathcal{M}$, i.e., exactly one of $\mathcal{M}[sd]$ and $\mathcal{M}[s'd]$ is an accepting state.

In the algorithm, the sets $\mathcal{S}_{\mathcal{T}}$ and $\mathcal{D}_{\mathcal{T}}$ are maintained in a binary discrimination tree, where each internal node is labeled by a string in $\mathcal{D}$ and each leaf is labeled by a string in $\mathcal{S}$.

In Figure 5.1, a DFA and its corresponding discrimination tree are illustrated. Both were taken from [31].


(a)

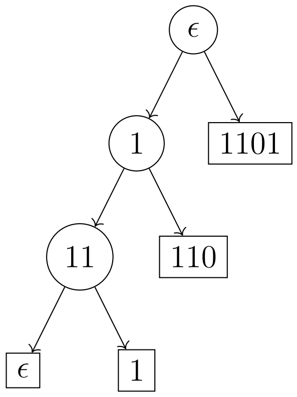

(b)

Figure 5.1: (a) DFA counting the number of 1's in the input 3 mod 4. (b) A discrimination tree for this automaton.

5.1.3 The Algorithm for Learning Finite Automata

In this section we describe each subroutine of the algorithm with the corresponding pseudocodes.

5.1.4 Initialization and Data Structures

The initialization of the algorithm is straightforward. A hypothesis automaton is constructed consisting in a single (accepting or rejecting depending on the result of **MQ**(ϵ)) state with self-loops for all the possible transitions ($\forall b \in \Sigma$). After that, an **EQ** is performed on this automaton and a counterexample φ is obtained. If the result of **MQ**(ϵ) was $\perp$, it means that ϵ is a rejecting state so it has to be the 0-child of the root node. If the result was $\top$, it means that ϵ is an accepting state so it has to be the 1-child of the root node. The discrimination tree $\mathcal{T}$ is initialized having the root labeled with the **distinguishing string** ϵ and two leaves with **access strings** ϵ and φ. In the following pseudocode it is assumed that a counterexample is obtained. Take into account that it is possible that $\mathcal{H} \equiv \mathcal{M}$.

Algorithm 13 Initialization routine

```
1: function KEARNS-AND-VAZIRANI-INIT
2:     $is\_accepting \leftarrow$ MQ ($\epsilon$)
3:     $\mathcal{H} \leftarrow$ CREATE-SINGLE-STATEDFA($is\_accepting$)
4:     $are\_eq, \varphi \leftarrow$ EQ ($\mathcal{H}, \mathcal{M}$)
5:     $r_{\mathcal{T}} \leftarrow$ ROOT-NODE($\epsilon$)
6:     $epsilon\_node \leftarrow$ NODE($\epsilon$)
7:     $counterexample\_node \leftarrow$ NODE($\varphi$)
8:     if $is\_accepting$ then
9:         $r_{\mathcal{T}}.right = epsilon\_node$
10:        $r_{\mathcal{T}}.left = counterexample\_node$
11:    else
12:        $r_{\mathcal{T}}.right = counterexample\_node$
13:        $r_{\mathcal{T}}.left = epsilon\_node$
14:    end if
15:    $\mathcal{T} \leftarrow$ INIT-TREE($r_{\mathcal{T}}$)
16:    return $are\_eq, \mathcal{H}, \mathcal{T}$
17: end function
```

5.1.5 Sift operation

The `Sift` subroutine takes as input a string s and the current discrimination tree $\mathcal{T}$, and outputs the access string in $\mathcal{T}$ of the equivalence class of $\mathcal{M}[s]$, the state of $\mathcal{M}$ accessed by s.

Starting at $r_{\mathcal{T}}$, if we are at an internal node labeled with a **distinguishing string** d, we make a **MQ** on the sequence sd and go to the left (0-child) or right subtree (1-child) as indicated by the query answer (left on reject, right on accept). We continue until we reach a leaf l, which it is the output of the operation.

Algorithm 14 Sift operation

```
1: function SIFT($s, \mathcal{T}$)
2:     $n \leftarrow$ ROOT-NODE()
3:     while $n \in \mathcal{D}_{\mathcal{T}}$ do
4:         $d \leftarrow n.discriminator$
5:         $o \leftarrow$ MQ $(sd)$
6:         $n \leftarrow n.children[o]$
7:     end while
8:     return $n$
9: end function
```

5.1.6 Tentative Hypothesis

The `Tentative-Hypothesis` operation describes the construction of a hypothesis automaton $\mathcal{H}$. The states of this automaton can be identified with the **access strings** in the discrimination tree. Once the states of the automaton are created, transitions have to be defined. For each state access string s, the transition $b \in \Sigma$ is created *sifting* the sequence sb.

Algorithm 15 Tentative-Hypothesis operation

```
 1: function TENTATIVE-HYPOTHESIS($\mathcal{T}$)
 2:     $\mathcal{H} \leftarrow$ CREATE-DFA()
 3:     $\mathcal{S}_{\mathcal{T}} \leftarrow$ GET-TREE-LEAVES($\mathcal{T}$)
 4:     for each $l \in \mathcal{S}_{\mathcal{T}}$ do
 5:         CREATE-STATE($\mathcal{H}, l$)
 6:     end for
 7:     for each $s \in \mathcal{H}$ do
 8:         for each $b \in \Sigma$ do
 9:             $s' \leftarrow$ SIFT($sb, \mathcal{T}$)
10:             CREATE-TRANSITION($\mathcal{H}, b, s, s'$)
11:         end for
12:     end for
13:     return $\mathcal{H}$
14: end function
```

5.1.7 Lowest common ancestor

The *lowest common ancestor* of two nodes in a binary tree is the shared ancestor that is located farthest from the root [33]. An efficient way of computing this operation is achieved if every node stores a pointer to their parent node, so that it has an average time complexity of $\mathcal{O}(\log n)$.

Algorithm 16 Lowest-Common-Ancestor operation

```
 1: function LOWEST-COMMON-ANCESTOR($a, b$)
 2:     if $a.depth < b.depth$ then
 3:         $temp \leftarrow a$
 4:         $a \leftarrow b$
 5:         $b \leftarrow temp$
 6:     end if
 7:     while $a.depth > b.depth$ do
 8:         $a \leftarrow a.parent$
 9:     end while
10:     while $a \neq b$ do
11:         $a \leftarrow a.parent$
12:         $b \leftarrow b.parent$
13:     end while
         return $a$
14: end function
```

5.1.8 Update Tree

When an **EQ** is made with $\mathcal{H}$, a counterexample φ can be obtained. This is the reason why the `Update-Tree` operation must be introduced.

The key concept here is understanding why φ was obtained as a counterexample and how the tree is changed. The operation finds a new **access string**, and updates $\mathcal{T}$ by adding a new leaf node labeled with the new access string.

This procedure first finds the string which makes $\mathcal{M}$ and $\mathcal{H}$ output different results. This is achieved by traversing a list of prefixes γ of φ and finding the first prefix for which $\textsc{Sift}(\gamma[i], \mathcal{T})$ is different from $\mathcal{H}[\gamma[i]]$, being $\gamma[i]$ the corresponding prefix.

Let s_i be $\textsc{Sift}(\gamma[i], \mathcal{T})$, $\hat{s}_i$ be $\mathcal{H}[\gamma[i]]$ and j the least i such that $s_i \neq \hat{s}_i$. The node labeled with the **access string** s_{j-1} has to be replaced with an internal node with two leaves. One is labeled with the **access string** s_{j-1} and the other with the **access string** $\gamma[j-1]$. The newly created internal node is labeled with the **distinguishing string** $\varphi_j d$, where d is the correct **distinguishing string** for s_j and $\hat{s}_j$, and can be obtained with the `Lowest-Common-Ancestor` operation.

Algorithm 17 Update-tree operation

```
 1: procedure UPDATE-TREE($\varphi, \mathcal{T}$)
 2:     $\gamma \leftarrow$ GET-PREFIXES($\varphi$)
 3:     for $i \leftarrow 0$ to $\gamma.length$ do
 4:         $s_i \leftarrow$ SIFT($\gamma[i], \mathcal{T}$)
 5:         $\hat{s}_i \leftarrow \mathcal{H}[\gamma[i]]$
 6:         if $s_i \neq \hat{s}_i$ then
 7:             $j \leftarrow i$
 8:             $d \leftarrow$ LOWEST-COMMON-ANCESTOR($s_j, \hat{s}_j$)
 9:             REPLACE-NODE($s_{j-1}, \gamma[j-1], d, \mathcal{T}$)
10:         end if
11:     end for
12: end procedure
```

5.1.9 Putting it all together

With the initialization, the `Tentative-Hypothesis` and the `Update-Tree` operations, the complete algorithm can be presented. We first start by performing the initialization routine.

Subsequently, the main loop begins. The function TENTATIVE-HYPOTHESIS($\mathcal{T}$) is invoked to obtain a new hypothesis $\mathcal{H}$. During each iteration, the tree is updated using UPDATE-TREE($\mathcal{T}, \varphi$). The loop continues until $\mathcal{H}$ and $\mathcal{M}$ are equivalent.

Algorithm 18 Kearns and Vazirani's Algorithm

```
 1: function KEARNS-AND-VAZIRANI
 2:     $are\_eq, \mathcal{H}, \mathcal{T} \leftarrow$ KEARNS-AND-VAZIRANI-INIT()
 3:     while $not(are\_eq)$ do
 4:         $\mathcal{H} \leftarrow$ TENTATIVE-HYPOTHESIS($\mathcal{T}$)
 5:         $(are\_eq, \varphi) \leftarrow$ EQ($\mathcal{H}, \mathcal{M}$)
 6:         if $are\_eq$ then
 7:             return $\mathcal{H}$
 8:         end if
 9:         UPDATE-TREE($\mathcal{T}, \varphi$)
10:         return $\mathcal{H}$
11:     end while
12: end function
```

The number of times the main loop of the algorithm is executed is exactly $size(\mathcal{M})$ because in each iteration a new state of $\mathcal{H}$ is discovered and the loop finishes when $\mathcal{H} \equiv \mathcal{M}$ [31].

Each of the loop executions requires $\mathcal{O}(size(\mathcal{M}) + n)$ sifting operations, where n is the length of the longest counterexample.

5.1.10 Example run

Now that the formal algorithm was introduced, we can see how an automaton $\mathcal{H}$ can be obtained running the algorithm with the automaton in figure Figure 5.1. First of all, a single state hypothesis automaton is constructed with self-loops for both the 0 and 1 transitions (Figure 5.2a). A **MQ** is performed with ϵ getting $\perp$ as result and thus the single state hypothesis automaton is a rejecting one.

An **EQ** is performed on this automaton, getting a counterexample 1101. The discrimination tree is initialized having the root node $r_{\mathcal{T}}$ labeled with the **distinguishing string** ϵ and two leaves labeled with **access strings** ϵ and 1101. As the result of the **MQ** of ϵ is $\perp$, ϵ is the left child of $r_{\mathcal{T}}$ and 1101 is the right child of $r_{\mathcal{T}}$.

Now the main loop starts. A new $\mathcal{H}$ is obtained after calling the `Tentative-`

`Hypothesis` function (Figure 5.2c). An **EQ** is performed on this automaton, getting the same counterexample 1101. The list of prefixes γ of this counterexample is $[1, 11, 110, 1101]$. For this stage of the algorithm remember that for each prefix $\gamma[i]$[1] of γ we have the following:

- $s_i \leftarrow \textsc{Sift}(\gamma[i], \mathcal{T})$.
- $\hat{s}_i \leftarrow \mathcal{H}[\gamma[i]]$.

If we execute the `Update Tree` procedure, we observe that the smallest index i for which $s_i \neq \hat{s}_i$ is $j = 4$, with the prefix 1101 since $s_i = 1101$ and $\hat{s}_i = \epsilon$. Following this, the node labeled with the **access string** s_{j-1}, which is ϵ, needs to be replaced. This node becomes one of the children of the new internal node. The other leaf is $\gamma[j-1]$, which is 110.

To determine the label of the newly created internal node the `Lowest-Common-Ancestor` operation is applied to the nodes ϵ and 1101, resulting in ϵ. This indicates that the concatenation of 1101_j (1) and ϵ, which is 1, serves as the label for the new internal node. Additionally, ϵ is the correct **distinguishing string** for 1101 and ϵ.

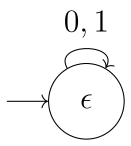
Finally, an **MQ** can be executed using either **1**ϵ or **1**110. Assuming it is performed with 1110, the output of the **MQ** is $\top$. Therefore, the 1-child of 1 is 1110, and the 0-child is ϵ.

A new $\mathcal{H}$ is obtained, but it is still not equal to $\mathcal{M}$, and the counterexample is once again 1101. Now, $s_i = 11$ and $\hat{s}_i = \epsilon$. The node to be replaced is labeled with the **access string** s_{j-1}, which is ϵ, and the other leaf is $\gamma[j-1]$, which is 1. The `Lowest-Common-Ancestor` operation is invoked with the nodes ϵ and 110, resulting in 1. This indicates that the concatenation of 1101_j (1) with 1 (11) is the label of the new internal node, and 1 is the correct **distinguishing string** for 110 and ϵ.

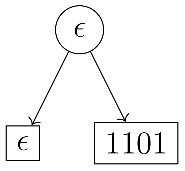
Finally, an **MQ** can be performed using either **11**ϵ or **11**1. Asumming it is with 111, the output of the **MQ** is $\top$, so the 1-child of 11 is 1 and the 0-child is ϵ.

The $\mathcal{H}$ obtained is equivalent to $\mathcal{M}$ so $\mathcal{H}$ is returned (Figure 5.2g) and the algorithm finishes.

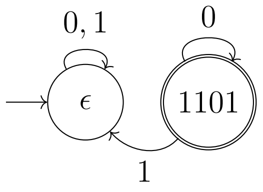
[1]With $1 \leq i \leq 4$.



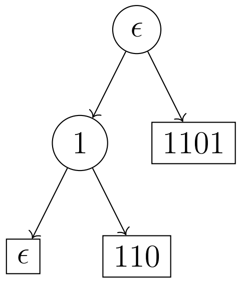
(a) Single state DFA



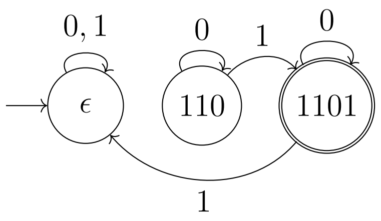
(b) Discrimination tree after initialization



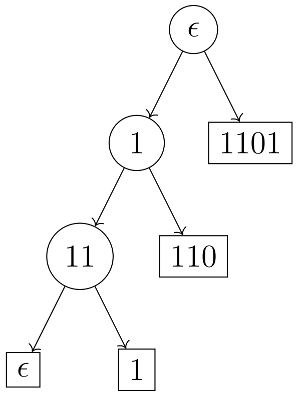
(c) Hypothesis after first counterexample



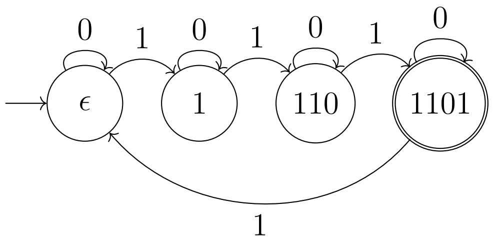
(d) Discrimination tree after second counterexample



(e) Hypothesis after second counterexample



(f) Discrimination tree after third counterexample



(g) Final hypothesis automaton

Figure 5.2: Evolution of hypothesis and discrimination tree during a run of Kearns and Vazirani's algorithm

5.1.11 Counterexample Exhaustion

One of the most expensive processes in terms of execution time for the algorithm is the **EQ**. In the example run, it can be observed that the counterexample given is always 1101. For this reason, we can optimize the algorithm by reusing the counterexample and avoiding the **EQ** until **MQ** (φ) is not equal to ACCEPTS(φ)[2]. This way, unnecessary **EQ** calls can be reduced, leading to improved efficiency in the algorithm's execution.

Algorithm 19 Kearns and Vazirani's optimized Algorithm

```
1: function KEARNS-AND-VAZIRANI
2:     $are\_eq, \mathcal{H}, \mathcal{T} \leftarrow$ KEARNS-AND-VAZIRANI-INIT()
3:     if $not(are\_eq)$ then
4:         $\mathcal{H} \leftarrow$ TENTATIVE-HYPOTHESIS($\mathcal{T}$)
5:         $are\_eq, \varphi \leftarrow$ EQ ($\mathcal{H}, \mathcal{M}$)
6:         while $not(are\_eq)$ do
7:             UPDATE-TREE($\mathcal{T}, \varphi$)
8:             $\mathcal{H} \leftarrow$ TENTATIVE-HYPOTHESIS($\mathcal{T}$)
9:             if MQ($\varphi$) = ACCEPTS($\varphi$) then
10:                $are\_eq, \varphi \leftarrow$ EQ ($\mathcal{H}, \mathcal{M}$)
11:            end if
12:        end while
13:    end if
14:    return $\mathcal{H}$
15: end function
```

5.1.12 Spanning-Tree Hypothesis

Kearns and Vazirani and other authors such as Rivest and Schapire [34] or Angluin [24] describe that the observation data structure is what is being built during the actual learning phase, and the hypothesis is then constructed in a separate step, from the information stored in the observation data structure [32].

Howar [35] proposed that rather than constantly reconstruct the hypothesis from scratch (a problem we have in the Kearns and Vazirani algorithm), it is more effective to use the hypothesis as a primary representation of the learner's knowledge [32].

[2]We call `Accepts` to the function that receives a counterexample and returns $\perp$ if the hypothesis $\mathcal{H}$ does not accept the counterexample and $\top$ if the counterexample is accepted.

5.2 Smart Counterexample Processing

In their paper, Rivest-Schapire [34] present a procedure to find a suffix e that exposes the difference between two rows in an Observation Table. The process of finding the desired suffix is now explained in detail and it is based on the explanation of Rick ten Tije [36].

We have two DFAs: $\mathcal{M}$, the target automaton we are learning, and $\mathcal{H}$, the hypothesis. Assuming we have a counterexample φ that distinguishes these two DFAs, we check if φ is accepted by $\mathcal{M}$.

For each iteration, we split φ into two parts. Let u be the first part of φ, and v be the remaining part. We then pass u through $\mathcal{H}$ and observe the state it ends in. Recall that each state has its own prefix representing the state. We take the prefix of that state, denoted as p. We append v to p and put it through $\mathcal{M}$, denoted as $\mathbf{MQ}(pv)$. One of the following two scenarios can occur:

- $\mathbf{MQ}(pv) = \mathbf{MQ}(\varphi)$, indicating that v or one of its suffixes distinguishes $\mathcal{M}$ and $\mathcal{H}$. To pinpoint the exact suffix, we proceed to split φ at index $u + \frac{1}{2}v$. This means that in the next iteration, u will have more symbols from φ and v will have fewer symbols.
- $\mathbf{MQ}(pv) \neq \mathbf{MQ}(\varphi)$, indicating that v and all of its suffixes do not distinguish $\mathcal{M}$ from $\mathcal{H}$. In this case, a part of u is needed to find the correct suffix. Thus, for the next iteration, we split φ at index $\frac{1}{2}u$.

The splitting process resembles a binary search. The first split of φ occurs at index $\frac{1}{2}\varphi$. The second split, depending on $\mathbf{MQ}(pv)$ and $\mathbf{MQ}(\varphi)$, will be at either $\frac{3}{4}\varphi$ or $\frac{1}{4}\varphi$. This entire process is repeated until we find the exact suffix v of φ for which a one-symbol longer suffix or one-symbol shorter suffix of φ changes the outcome of $\mathbf{MQ}(pv) = \mathbf{MQ}(\varphi)$ to $\mathbf{MQ}(pv) \neq \mathbf{MQ}(\varphi)$ or vice versa. Due to the binary search approach, we require $\log(\varphi)$ $\mathbf{MQ}$s to find the desired suffix.

Algorithm 20 Rivest - Schapire counterexample processing

```
function RIVEST-SCHAPIRE($\varphi, \mathcal{H}$)
    $cex\_out \leftarrow$ MQ($\varphi$)
    $lower \leftarrow 1$
    $upper \leftarrow \varphi.length - 2$
    while $True$ do
        $mid \leftarrow (lower + upper)//2$                ▷ // denotes Integer division
        $u = \varphi[: mid]$
        $final\_state \leftarrow \mathcal{H}.getFinalState(u)$
        $p \leftarrow final\_state.prefix$
        $v \leftarrow \varphi[mid :]$
        $mq \leftarrow$ MQ($pv$)
        if $mq = cex\_out$ then
            $lower \leftarrow mid + 1$
            if $upper < lower$ then
                return $v[1 :]$
            end if
        else
            $upper \leftarrow mid - 1$
            if $upper < lower$ then
                return $v$
            end if
        end if
    end while
end function
```

5.3 Observation Pack Algorithm

The Observation Pack algorithm by Howar is a combination of the discrimination tree data structure with the counterexample analysis proposed by Rivest and Schapire. In this section we explain in detail the algorithm based on the presentation made by Isberner [32]. We refer to the referenced work as Isberner's thesis.

5.3.1 Initialization and Data Structures

First of all we have to create a new single-state hypothesis and a discrimination tree consisting of an inner node (labeled with ϵ) and two *nil* leaves, and determine the leaf corresponding to the initial state by *sifting* ϵ into the tree. If the leaf is in the 1-subtree of the root it means that now the right node of the root is labeled with ϵ and corresponds to the initial state. If the leaf is in the 0-subtree it means that now the left node of the root is labeled with ϵ and corresponds to the initial state.

For this reason we have to make a change in the `Sift` operation, because we have to handle the case in which a nil leaf is reached. This new version is specified in the following algorithm.

Algorithm 21 New version of the Sift operation

```
1:  function SIFT($s, \mathcal{T}$)
2:      $n \leftarrow$ ROOT-NODE()
3:      while $n \in \mathcal{D}_{\mathcal{T}}$ do
4:          $d \leftarrow n.discriminator$
5:          $o \leftarrow$ MQ $(sd)$
6:          $n \leftarrow n.children[o]$
7:      end while
8:      if $n = nil$ then
9:          CREATE-NODE($s$)
10:     end if
11:     return $n$
12: end function
```

Something important to mention is that Isberner points out that *"the representative prefix associated with a state is called access sequence, and it can be obtained by concatenating all transition labels on the path from the root of the spanning tree to the respective state"*. This will be referred to as *aseq*.

Subsequently, we have to create a link between the leaf l and the created node q. This is refered as the LINK(l, q) operation. Every state q has a pointer to its corresponding node in the discrimination tree (referred as *q.node*), and conversely, every leaf has a pointer to the state it corresponds to (referred as *l.state*, which may be *nil*).

Finally, before concluding the initialization of the algorithm we have to take a look to the CLOSE-TRANSITIONS($\mathcal{H}, \mathcal{T}$) operation (Algorithm 22). We assume that the outgoing transitions of every state $q \in \mathcal{H}$ can be accessed as $q.trans[a]$ for $a \in \Sigma$. Every outgoing transition t can either be a tree or non-tree transition.

Non-tree transitions point to the root node of the discrimination tree[3]. For a non-tree transition, this target node is referred to via *t.tgt_node*. On the other hand, for a tree transition t, *t.tgt_node* refers to the leaf associated with its target state. The target state of a transition t is referred as *t.tgt_state*.

As the *t.tgt_node* of a non-tree transition is not a leaf, the *t.tgt_state* will be *nil*. This is referred as an open transition and $Open(\mathcal{H})$ denotes the set of all open transitions in $\mathcal{H}$.

Algorithm 22 Close transition procedure

```
1:  procedure CLOSE-TRANSITIONS($\mathcal{H}, \mathcal{T}$)
2:      while $Open(\mathcal{H}) \neq \emptyset$ do
3:          $t \leftarrow choose(Open(\mathcal{H}))$
4:          $tgt \leftarrow$ SIFT($t.aseq$)
5:          $t.tgt\_node \leftarrow tgt$
6:          if $t.tgt\_node = nil$ then
7:              $q \leftarrow$ CREATE-STATE($t$)
8:              LINK($t.tgt\_node, q$)
9:          end if
10:     end while
11: end procedure
```

Algorithm 23 Observation Pack Initialization routine

```
1: function OBSERVATIONPACK-INIT
2:     $\mathcal{H} \leftarrow$ CREATE-HYPOTHESIS()  ▷ create new hypothesis with single state $\epsilon$
3:     $\mathcal{T} \leftarrow$ MAKE-INNER($\epsilon, nil, nil$)
4:     $l \leftarrow$ SIFT($\epsilon$)
5:     LINK($l, \epsilon, \mathcal{H}$)
6:     CLOSE-TRANSITIONS($\mathcal{H}, \mathcal{T}$)
7:     return $\mathcal{H}, \mathcal{T}$
8: end function
```

5.3.2 Refinement

Now comes the part where a suffix-based counterexample analysis (Rivest Schapire) is introduced in order to refine the hypothesis and discrimination tree. The corresponding pseudocode is shown as Algorithm 24.

[3]This is a change from Isberner's thesis, where he specifies that non-tree transitions point to nodes in the discrimination tree.

The idea is that a counterexample φ can be decomposed into $\varphi = (\hat{u}, \hat{a}, \hat{v})$ with the following property: $q_{pred} = \mathcal{H}[\hat{u}]$ and $q_{old} = q_{pred}.trans[\hat{a}]$, we then have $\mathbf{MQ}(q_{pred}.aseq, \hat{a}, \hat{v}]) \neq \mathbf{MQ}(q_{old}.aseq, \hat{v})$. Thus, the $\hat{a}$-successor of q_{pred} must be different from q_{old}, which calls the introduction of a new state q_{new}.

It is important to note that all the incoming and outgoing transitions of q_{old} must be reset. Moreover, the link between q_{old} and the leaf that formerly corresponded to q_{old} in the tree must be removed. This leaf is replaced by an inner node with discriminator $\hat{v}$ and two leaves. The states q_{old} and q_{new} are then linked to these leaves, according to their future behaviour after doing $\mathbf{MQ}(q_{old}.aseq, \hat{v})$.

Finally, the open transitions in $\mathcal{H}$ are closed concluding the refinement of $\mathcal{H}$ and $\mathcal{T}$.

Algorithm 24 Realization of refinement in the Observation Pack algorithm

```
1: procedure ObservationPack-Refine($\varphi$)
2:     $\hat{u}, \hat{a}, \hat{v} \leftarrow$ Analyze-OutIncons($\varphi$)
3:     Split($\mathcal{H}, \mathcal{T}, \hat{u}, \hat{a}, \hat{v}$)
4:     Close-Transitions($\mathcal{H}, \mathcal{T}$)
5: end procedure
1: procedure Split($\mathcal{H}, \mathcal{T}, \hat{u}, \hat{a}, \hat{v}$)
2:     $q_{pred} \leftarrow \mathcal{H}[\hat{u}]$
3:     $q_{old} \leftarrow q_{pred}.trans[\hat{a}]$
4:     $q_{new} \leftarrow$ Create-State($q_{old}.aseq$)
5:     $l_0, l_1 \leftarrow$ Split-Leaf($q_{old}.node, \hat{v}$)
6:     if MQ($q_{old}.aseq, \hat{v}$) $= \perp$ then
7:         Link($l_0, q_{old}$)
8:         Link($l_1, q_{new}$)
9:     else
10:        Link($l_0, q_{new}$)
11:        Link($l_1, q_{old}$)
12:    end if
13: end procedure
```

5.3.3 Putting it all together

With the initialization and refinement algorithms the complete algorithm can be presented. We first start by initializing $\mathcal{H}$ and $\mathcal{T}$. Subsequently, until $\mathcal{H}$ and $\mathcal{M}$ are equivalent, $\mathcal{H}$ and $\mathcal{T}$ are refined. It can happen that the counterexample is the same as before or that an **EQ** has to be executed to get a new counterexample.

Algorithm 25 Observation Pack Algorithm

```
1: function ObservationPack
2:     $\mathcal{H}, \mathcal{T} \leftarrow$ ObservationPack-Init()
3:     $(are\_eq, \varphi) \leftarrow \mathbf{EQ}(\mathcal{H}, \mathcal{M})$
4:     while $not(are\_eq)$ do
5:         ObservationPack-Refine()
6:         if $\mathbf{MQ}(\varphi) =$ Accepts$(\varphi)$ then
7:             $(are\_eq, \varphi) \leftarrow \mathbf{EQ}(\mathcal{H}, \mathcal{M})$
8:         end if
9:     end while
10:    return $\mathcal{H}$
11: end function
```

In Table 5.1, the worst-case complexity of Observation Pack algorithm is reported defining first some parameters [32]:

- n = size of target DFA $\mathcal{M}$.
- k = size of input alphabet.
- m = length of the longest counterexample returned by an equivalence query.

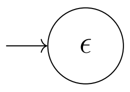
Query Complexity	**Symbol Complexity**	**Space Complexity**
$\mathcal{O}(kn^2 + n \log m)$	$\mathcal{O}(kn^2m + nm \log m)$	$\mathcal{O}(kn + nm)$

Table 5.1: Worst-case complexity of the Observation Pack algorithm

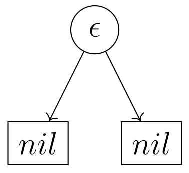
5.3.4 Example Run

Now that the complete algorithm was introduced, we can see how an automaton $\mathcal{H}$ can be obtained running the algorithm with the automaton in figure Figure 5.1.

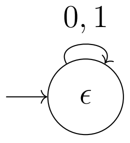
First of all, the spanning-tree hypothesis is initialized as a single state DFA with the access sequence ϵ (Figure 5.3a) and the discrimination tree with an ϵ labeled root node with two *nil* leaves (Figure 5.3b) . In second place, Sift(ϵ) is executed, creating a new leaf labeled with ϵ in the 0-subtree (Figure 5.3d). This means that the single DFA state is a rejecting one. During initialization, both symbols 0 and 1 are found to lead to the ϵ state (Figure 5.3c).



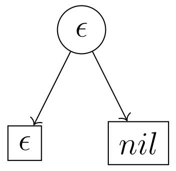
(a) Single state DFA



(b) Discrimination tree with nil leaves



(c) Single rejecting state DFA

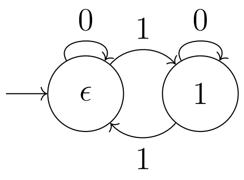


(d) Discrimination tree with ϵ leave

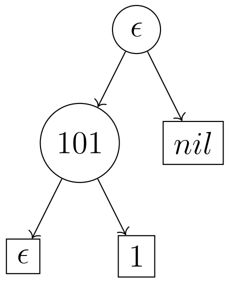
Figure 5.3: Initialization of hypothesis and discrimination tree during a run of Observation Pack

The initial hypothesis $\mathcal{H}$ classifies some sequences incorrectly. One of these sequences is $\varphi = 1101$, since $\textsc{Accepts}(\varphi) = \bot$ but $\mathbf{MQ}(\varphi) = \top$. Applying suffix-based counterexample analysis, a decomposition $(\hat{u}, \hat{a}, \hat{v}) = (\epsilon, 1, 101)$ is determined. Thus, the 1-transition of $q_{pred} = \mathcal{H}[\epsilon] = \epsilon$ is converted into a tree transition, resulting in the introduction of a new state. Furthermore, the leaf in the discrimination tree corresponding to $q_{old} = \mathcal{H}[1] = \epsilon$ is split, using 101 as the discriminator. The resulting data structures, after closing all transitions are shown in Figure 5.4a and Figure 5.4b

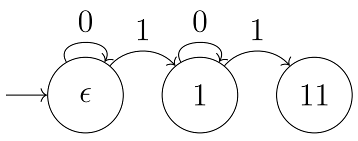
This new refined hypothesis still classifies φ incorrectly. A second suffix-based counterexample analysis returns the decomposition $(\hat{u}, \hat{a}, \hat{v}) = (1, 1, 01)$. Converting the 1-transition of $q_{pred} = \mathcal{H}[1] = 1$ into a tree transition, and splitting the leaf associated with $q_{old} = \mathcal{H}[11] = \epsilon$ using 01 as discriminator results in the discrimination tree shown in Figure 5.4d. If we close transitions without taking into account the transitions of the new state 11, it will result in the DFA shown in figure Figure 5.4c. The 1-transition of this state will result in the introduction of a new state 111, which will be an accepting one (inferred from the sift operation). Closing the rest of transitions results in the final hypothesis shown in Figure 5.4e and the final discrimination tree shown in Figure 5.4f.



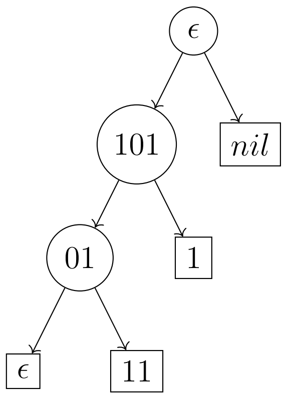
(a) Hypothesis after first refinement



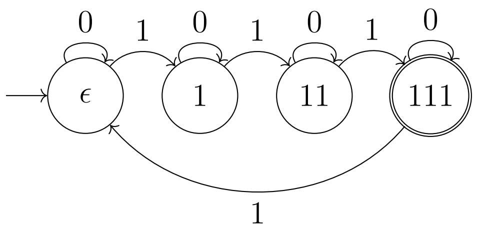
(b) Discrimination tree after first refinement



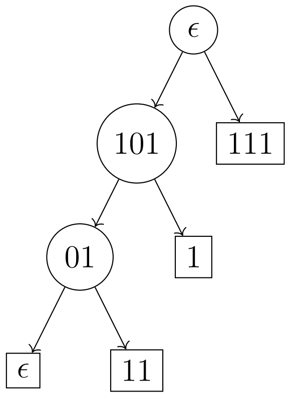
(c) Hypothesis after discovering the new state 11



(d) Discrimination tree after discovering the new state 11



(e) Hypothesis after second refinement



(f) Discrimination tree after second refinement

Figure 5.4: Evolution of hypothesis and discrimination tree during a run of Observation Pack

5.4 Benchmarks

The Kearns and Vazirani and Observation Pack algorithms are implemented in the Neural-Checker tool. As it is shown in this section, these algorithms use significantly more **EQ** s than the Observation Table algorithms. Therefore, the portions where they consume additional time are primarily in the **EQ** phase.

The Kearns and Vazirani algorithm was already implemented in the tool. Our goal was to optimize it because its runtime was significantly longer compared to the L^* algorithm. As a result of this, we implemented the counterexample exhaustion techique. Our investigation led us to discover the Observation Pack algorithm, which employs a smart counterexample processing method. We proceeded to implement it and conducted a comparison based on the following factors:

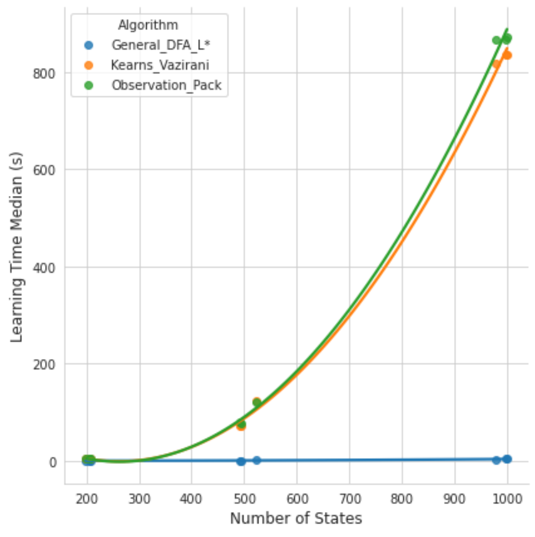
- Runtime duration.
- Number of model states.
- Alphabet size.

We conducted two experiments utilizing the BFS oracle (section 6.2) for the **EQ** operation. For a deeper understanding of oracles, please refer to Chapter 6. This will be crucial for comprehending the results of the experiments.

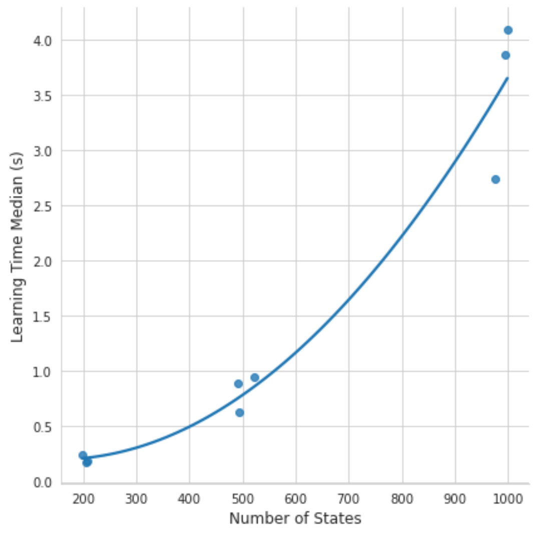
5.4.1 Experiment 1

In this experiment we compared Observation Pack, Kearns and Vazirani and L^*. For this, 9 random DFAs over a binary alphabet of nominal sizes 200, 500 and 1000 were generated, and each algorithm was run 9 times for each DFA. Figure 5.5a shows the learning time medians for every actual size. Observation Pack and Kearns and Vazirani execution time grows much faster than L^*. As the difference is so noticeable, Figure 5.5b shows that for 1000 states the running time of L^* is approximately 4 seconds, whereas for the other two algorithms, it exceeds 800 seconds.

We looked more closely at the observed differences, and it was not surprising to find that L^* uses fewer **EQ**s than the tree algorithms, as illustrated in Figure 5.6a. In contrast, the tree algorithms having fewer **MQ**, as shown in Figure 5.6b, was also as expected.

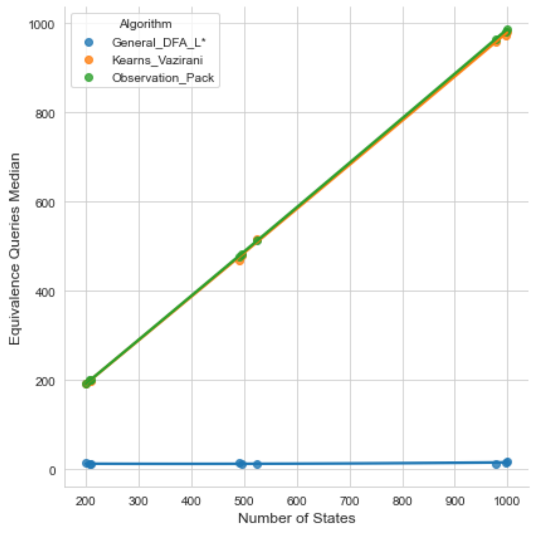


(a) Tree algorithms and L*.

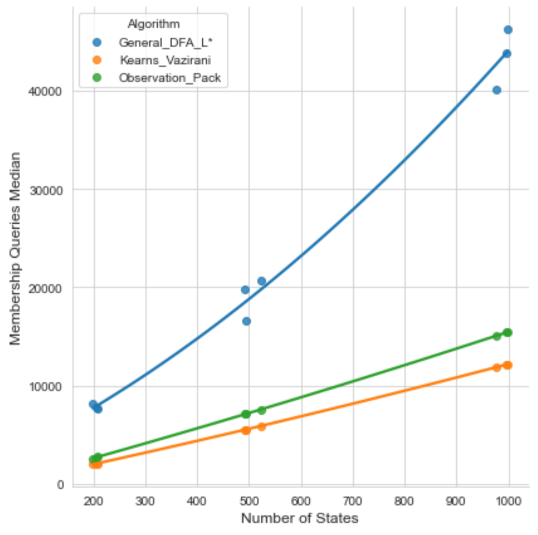


(b) L*.

Figure 5.5: Experiment 1 execution time.



(a) **EQ**s count.



(b) **MQ**s count.

Figure 5.6: Experiment 1 membership queries and equivalence queries.

5.4.2 Experiment 2

In this experiment 9 random DFA of nominal size 100 were generated for alphabet sizes 2, 32 and 64. We compared Observation Pack, Kearns and Vazirani and

L*. Each algorithm was run 9 times for each DFA. Figure 5.7 shows the learning time medians for every alphabet size. As it can be seen, once again the difference between the algorithms is noticeable.

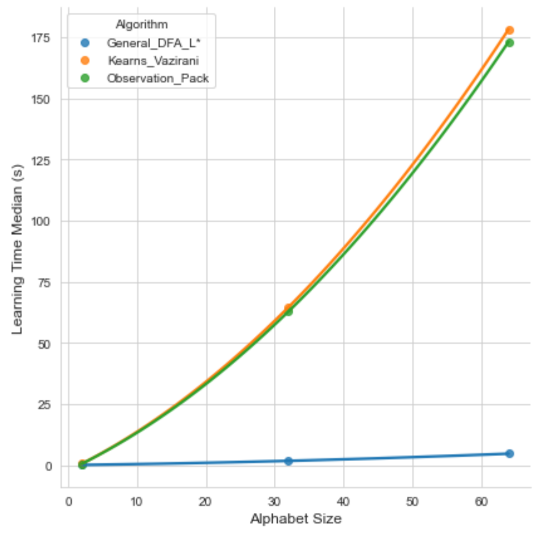


Figure 5.7: Execution time of Observation Pack, Kearns and Vazirani, and L* algorithms with different alphabet sizes.

5.5 Profiling

As mentioned earlier, the Observation Pack and Kearns and Vazirani's algorithms uses more **EQ**s than L* as the number of states increases. This is the primary motivation behind conducting profiling to identify the specific function responsible for consuming significant time, leading to differences in running time.

We used the `cProfile`[4] profiler from the Python standard library. A random DFA with a nominal size of 500 was generated using a binary alphabet for the analysis of the Observation Pack algorithm. Figure 5.8 illustrates that the algorithm required 252 seconds, with the **EQ** operation specifically taking 251 seconds.

This challenge is something we need to address in the future because it is making our algorithm less efficient when working with time consuming oracles. Our algorithm involves a lot of **EQ**, which are closely tied to the oracle. If these queries take too much time, it is affecting how long our entire process takes, and we are already seeing that impact.

[4]https://docs.python.org/3/library/profile.html

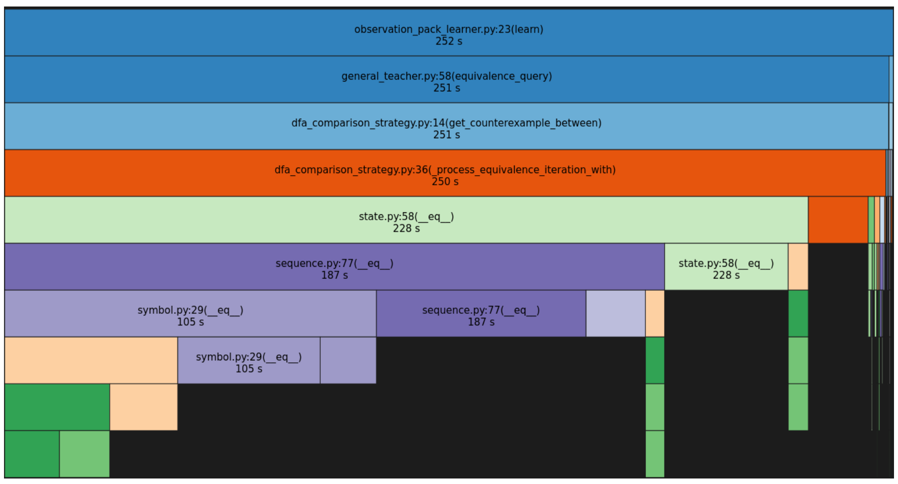


Figure 5.8: Observation Pack profiling.

We suspect that the issue might be related to abstractions in Python. To investigate, we conducted an experiment comparing the performance of a simple operation, summing two numbers, with and without the use of abstractions in the code. Two functions are defined for direct and abstracted sums. The experiment involves iterating through a total of 200 million iterations, measuring the execution time at intervals of 20 million iterations. The results, illustrated in Figure 5.9, reveal that the abstracted version consistently takes longer within each interval.

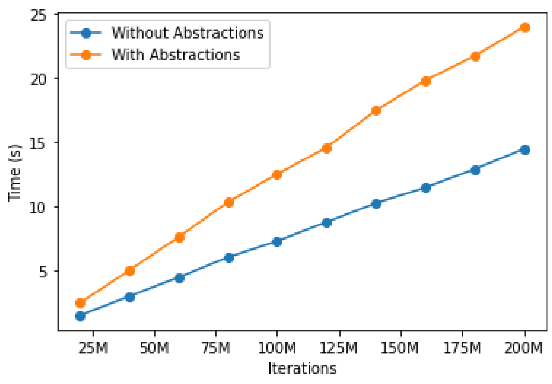


Figure 5.9: Summing two numbers with and without abstractions.

One potential solution involves a deeper investigation to optimize the implementation of the oracle used or to find other ones.

Another more challenging option discussed is to migrate the codebase to another language that offers faster performance, such as Golang. However, this poses difficulties as it would require not only migrating all the extraction algorithms but also all the automata and other implementations included in the used tool.

6 Oracles & Equivalences

An Oracle is just some abstract machine that knows the target and answers some queries [12]. In active learning algorithms, the teacher uses an oracle to process the **EQ**. In this chapter, some of the oracles that are used for model extractions are presented. These oracles are implemented in the Neural-Checker tool.

6.1 Hopcroft-Karp

This algorithm [18] defines an equivalence between two finite automatons using a set merging method, and can be used as an oracle for inference algorithms.

This equivalence algorithm has some limitations, both, the extracted model and the target models must be finite automatons; and can be harmful for the performance on inference algorithms that use lots of **EQ**s.

A positive aspect of this algorithm is that it can compare a DFA with a NFA (Non-Deterministic Finite Automaton) [12].

6.2 BFS Comparison Strategy

The BFS Comparison Strategy is an algorithm initially designed for DFA that our team extended its implementation to cover Moore and Mealy Machines.

The algorithm uses a breadth-first search strategy to explore pairs of states from two automatons and check if there is a pair with different final states. If such a pair is found, it returns a counterexample sequence. If no such pair is found, it concludes that the two automatons are equivalent.

6.3 Probably Approximately Correct learning

The active black-box model extraction algorithms introduced require some considerations when working with ANNs as there is no direct way of computing an **EQ**, and there is no termination guarantees, as the target languages may be more complex than automata [37]. In this scenario we need to use a sampling technique.

6.3.1 PAC-learning setting for languages

To understand PAC-learning for ANN, we first need to describe the setting for languages, as explained by Mayr and Yovine [4].

Let $\mathcal{D}$ be an unknown distribution over Σ^*, and the languages $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$. We call the symmetric difference between $\mathcal{L}_1$ and $\mathcal{L}_2$ to the set of sequences that only belong to one of the languages and is denoted as $\mathcal{L}_1 \oplus \mathcal{L}_2$. Formally, the symmetric difference is defined as the following: $\mathcal{L}_1 \oplus \mathcal{L}_2 = (\mathcal{L}_1 \setminus \mathcal{L}_2) \cup (\mathcal{L}_2 \setminus \mathcal{L}_1)$

$\mathbf{P}_{\mathcal{D}}(\mathcal{L}_1 \oplus \mathcal{L}_2)$ is defined as the probability that a sequence (chosen following the distribution $\mathcal{D}$) belongs to the symmetric difference. This could also be defined as the prediction error of $\mathcal{L}_1$ with respect to $\mathcal{L}_2$. $\mathcal{L}_1$ is ϵ-approximately correct with respect to $\mathcal{L}_2$ if $\mathbf{P}_{\mathcal{D}}(\mathcal{L}_1 \oplus \mathcal{L}_2) < \epsilon$ with $\epsilon \in (0, 1)$.

The oracle $\mathbf{EX}_{\mathcal{D}}(\mathcal{L}_1)$ draws an example sequence $x \in \Sigma^*$ following distribution $\mathcal{D}$ and tags it as $\top$ or $\bot$ according to whether it belongs to $\mathcal{L}_1$ or not. Each call to $\mathbf{EX}_{\mathcal{D}}$ is independent of each other.

If a PAC-Learning algorithm terminates, it outputs, with at least a probability of $1 - \delta$, a language that is ϵ-approximately correct with respect to a target language. The approximation ϵ, the confidence δ, the target language $\mathcal{L}_t$ and the oracle $\mathbf{EX}_{\mathcal{D}}(\mathcal{L}_t)$ are the input parameters of the algorithm with $\epsilon, \delta \in (0, 1)$.

A PAC-learning algorithm can incorporate an *approximate* equivalence test **EQ** which, in this setting, is defined as: using a sufficiently large sample of tagged sequences S generated by $\mathbf{EX}_{\mathcal{D}}(\mathcal{L}_t)$, checks a candidate output $\mathcal{L}_o$ against the target language $\mathcal{L}_t$. If the sample is such that $\forall x \in S$, $x \notin \mathcal{L}_o \oplus \mathcal{L}_t$, the algorithm successfully stops and outputs $\mathcal{L}_o$. Otherwise, it means that $S \cap (\mathcal{L}_o \oplus \mathcal{L}_t)$ is not empty, so the algorithm picks any sequence from the intersection as a counterexample and continues. The algorithm may also be allowed to call directly a **MQ**.

6.3.2 PAC-learning for ANN

Mayr and Yovine [4] asks whether it is possible to: given an ANN $\mathcal{N}$, build a DFA such that its language is ϵ-approximately correct with respect to $\mathcal{L}(\mathcal{N})$.

An active learning algorithm can be used to build this DFA, defining **MQ** as querying $\mathcal{N}$ itself, and **EQ** as checking whether $\mathcal{N}$ and the hypothesis automaton completely agrees in a sample set of sequences S_i, as defined earlier.

It is demonstrated [4] that if L^* terminates, it outputs a DFA $\mathcal{A}$ in which $\mathcal{L}(\mathcal{A})$ is ϵ-approximately correct with respect to $\mathcal{L}(\mathcal{N})$ with probability at least $1 - \delta$. Moreover, if $\mathcal{L}(\mathcal{N})$ is a regular language, L^* is proven to terminate.

6.4 Dataset Driven

Another approach for the oracles could be using a fixed dataset. An **EQ** is successful if all the sequences in the fixed dataset have the same result in the target model and in the model to check equivalence.

This could be useful when we want to make sure the result model algorithm is equal to the target model inside a subset of Σ^*.

This technique can also be combined with PAC, so the samples that PAC uses come from the dataset.

6.5 Random Walk

A random walk is known as a random process which describes a path including a succession of random steps in the mathematical space [38]. It has increasingly been popular in various disciplines such as mathematics and computer science.

Random walks can be used to analyze and simulate the randomness of objects and calculate the correlation among objects, which are useful in solving practical problems.

More specifically, when using Random Walk as an **EQ** technique, random symbols from Σ are sampled and a test is made to see if both models have the same result. To do this we have to specify two important parameters, *steps* and *reset probability*. The steps is the total amount of symbols that are concatenated, while the reset probability is the probability to substitute the accumulated sequence with the empty sequence. In every step, a weighted coin is flipped to determine whether to continue or restart. Then, a symbol is selected using a uniform distribution over Σ. It is important to mention that when a restart occurs, the step count is not reset, only a new sequence is initiated.

When choosing the reset probability there is a trade-off between the amount and the average length of the sequences that will be tested. Using a larger reset probability will end up in testing with shorter sequences, while a lower reset probability will end up testing with larger but fewer sequences.

It is important to note that while using a large amount of steps the algorithm ends up covering a larger portion of Σ^* but it takes more time. And in algorithms where a lot of **EQ**s are made (such as Kearns and Vazirani or Observation Pack) it could be time consuming.

6.6 State Prefix Random Walk

The problem with Random Walk is the fact that it is random and we do not have control over the sequences that get used for the **EQ**. This sometimes results in states that do not get explored. As it was discovered in [39], the State Prefix Random Walk oracle was introduced to solve this problem. This oracle is similar to Random Walk but instead of performing only one random walk, it conducts a random walk for every state the automaton has. To do this, the *access sequence* of every state is needed to use it as a suffix for every random walk process made.

The State Prefix Random Walk costs more than the normal Random Walk but it does a better job exploring the automaton. It makes sure that the **EQ** checks at least once every state.

6.7 Sampling techniques

Some **EQ** techniques (such as PAC) need to define a distribution over sequences to sample from. The following cases are evaluated: *uniform length*, *sampling from length distribution in a validation data* and *uniform word*.

Uniform length and **sampling from given length distribution** samplings are similar. Both techniques consist in sampling a word length. The first one does this with a uniform distribution between a minimum and maximum length while the latter does this sampling over a given distribution. Then, both techniques create every word with the sampled lengths choosing every symbol from an uniform distribution over Σ.

In the **uniform word** case, the sample is drawn from a uniform distribution over all possible words in Σ^* within specified length bounds.

6.8 Benchmarks

In the following experiments, two objectives are pursued: to measure the performance of the oracles mentioned in this chapter and analyze their relationship with the learning algorithm, and to test the **EQ** techniques that do not guarantee an equivalent automaton, such as Random Walk, State Prefix Random Walk or PAC. To test the latter, the BFS Comparison Strategy is used against the target automaton.

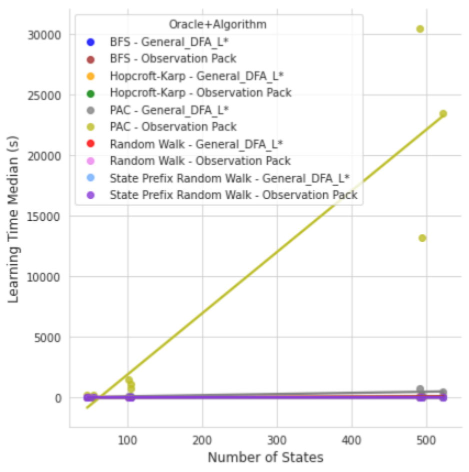
6.8.1 Experiment 1

For this experiment, automatons of 100, 300 and 500 states with a binary alphabet were generated and the runs were repeated three times over these generated DFAs. The algorithms used for the learning were General L* and Observation Pack. The oracles used for the process were Hopcroft-Karp, PAC, BFS Comparison Strategy, Random Walk and State Prefix Random Walk.

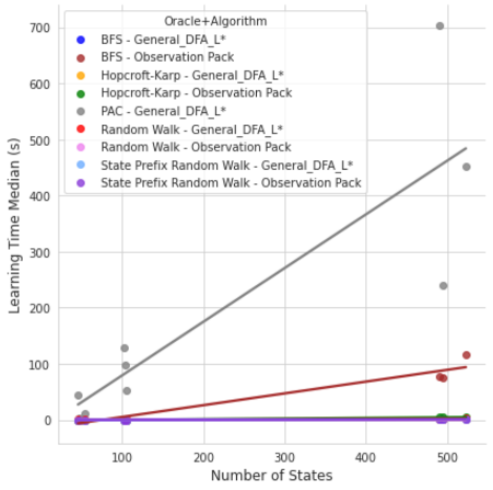
In Figure 6.1a and Figure 6.1b, it can be seen that PAC is an oracle with poor performance, specially with Observation Pack, being practically unusable.

Moreover, Observation Pack gets highly influenced by the oracle used, as previously stated on Chapter 5. This is a consequence of the amount of **EQ**s that the algorithm needs to perform the learning. L* theoretically is a less effective learning algorithm but empirically (as seen in Figure 6.1)it performs better than tree algorithms when using a "slow" oracle such as BFS Comparison Strategy Figure 6.1c. When using a "fast" oracle such as Random Walk we can safely state that Observation Pack performs better than L*.

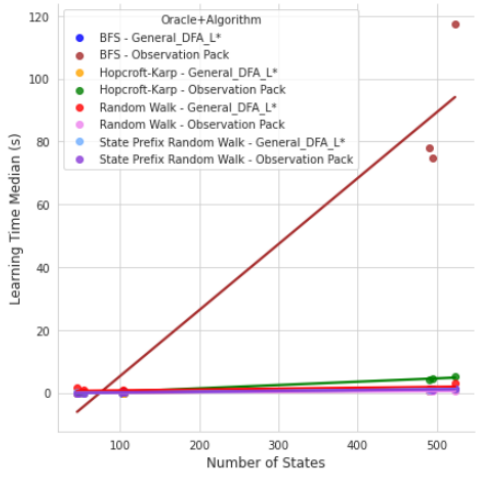
Finally, in Figure 6.1d Hopcroft-Karp shows some interesting behaviour since, unexpectedly, L* with Hopcroft-Karp is similar than oracles that do not assure an equivalent result on automatons such as Random Walk or State Prefix. However, it can be observed in Figure 6.1d that Random Walk sometimes incurs significant costs due to the length of the counterexamples, leading to an increase in the median time.



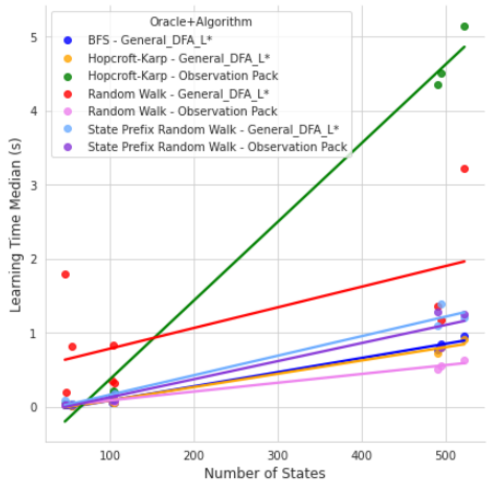
(a) Experiment 1 with all oracles.



(b) Experiment 1 without PAC - Observation Pack.



(c) Experiment 1(b) without PAC - General DFA L*.



(d) Experiment 1(c) without BFS - Observation Pack.

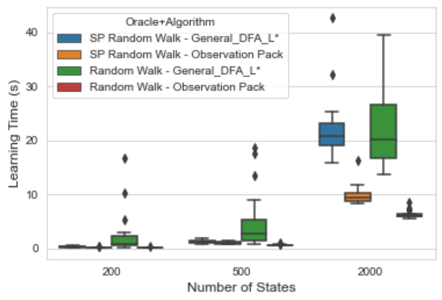
Figure 6.1: Experiment 1 results.

6.8.2 Experiment 2

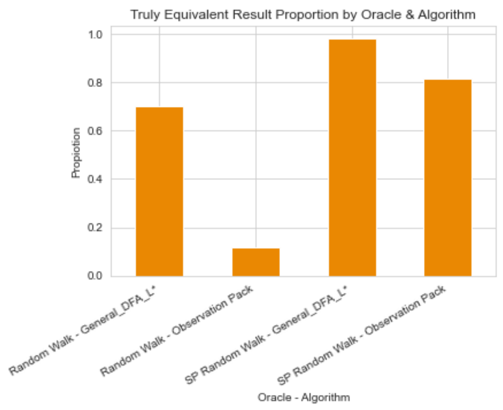
Here, we aim to compare Random Walk with State Prefix Random Walk (SP Random Walk) using L* and Observation Pack as the learning algorithms. DFAs with 200, 500 and 2000 states with a binary alphabet were generated.

The results in Figure 6.2 show that State Prefix Random Walk is a much better oracle overall; since it finishes the learning process in less time and outperforms Random Walk when comparing the result model.

Additionally, it can be observed in Figure 6.2a that L* has more variance in the learning time than Observation Pack. Results illustrated in Figure 6.2b could be affected by the latter. This should be further explored in future work.



(a) Experiment 2 learning time.



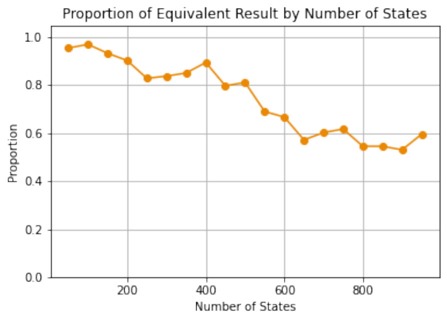
(b) Experiment 2 truly equivalent result.

Figure 6.2: Experiment 2 results.

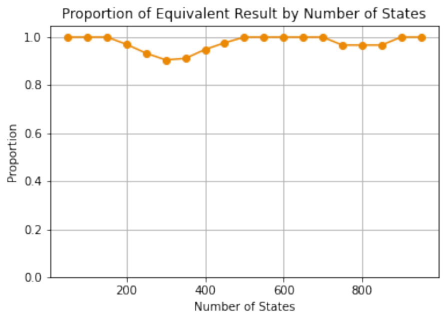
6.8.3 Experiment 3

In this experiment it is analyzed how the size of the automaton affects the learning result. 156 DFAs of different sizes between 100 and 1000 states with a binary alphabet were generated. The selected oracles were Random Walk with 3000 steps and 0.001 of reset probability and State Prefix Random Walk with 50 steps per state.

The results shows that as the amount of states grows, the learning with Random Walk becomes less precise. This could be explained because the Random Walk steps does not change as the complexity of the automaton increases. Furthermore, this issue does not seem to affect the State Prefix Random Walk as expected.



(a) Experiment 3 Random Walk.



(b) Experiment 3 State Prefix Random Walk.

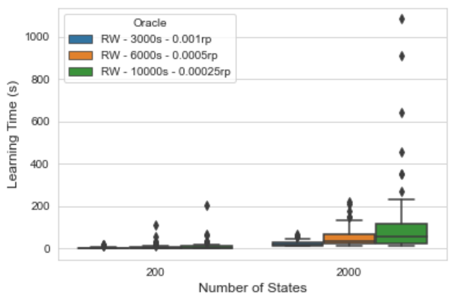
Figure 6.3: Experiment 3 results.

6.8.4 Experiment 4

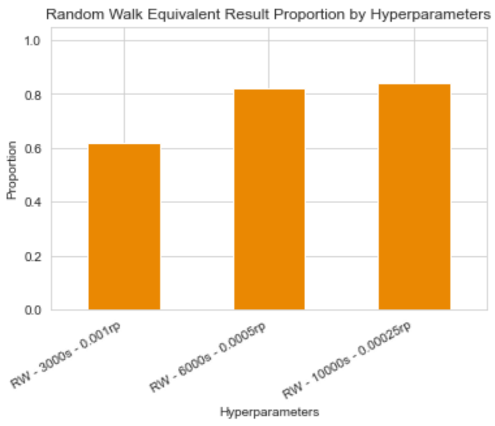
Experiment 4 shows how the Random Walk and State Prefix hyper-parameters affect the learning result and time. 100 DFAs of 200 and 2000 states with a binary alphabet were generated, and L* was used as the learning algorithm. The parameters for Random Walk were 3000, 6000 and 10000 *steps* with 0.001, 0.0005, 0.00025 of *reset probability* respectively. For State Prefix, the parameters were 2, 10 and 50 steps per state.

The results in Figure 6.4b show that, as expected, as the number of steps grows and the reset probability decreases, the probability of not reaching an equivalent model using Random Walk falls. However, this comes with a downside, as seen in Figure 6.4a, the duration of the learning process increases. As the number of steps increases and the reset probability decreases, the variance of the learning time also increases.

Moreover, the results in Figure 6.5b show that State Prefix oracle behaves in a similar way. It can be observed that as we increase the amount of steps, the accuracy increases. From 50 steps onwards, the learning results are almost always equivalent to the target automaton. In Figure 6.5a it can be seen that, in contrast to Random Walk, the duration while using State Prefix only decreases when using two steps.

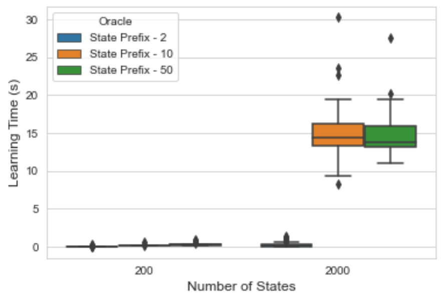


(a) Experiment 4 learning time.

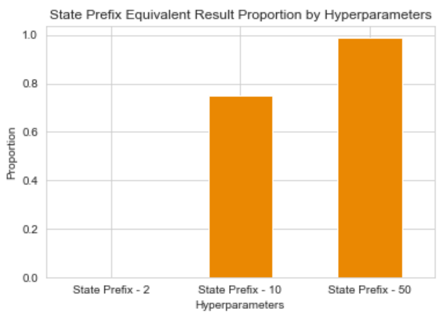


(b) Experiment 4 equivalent result.

Figure 6.4: Experiment 4 Random Walk results.



(a) Experiment 4 learning time.



(b) Experiment 4 equivalent result.

Figure 6.5: Experiment 4 State Prefix results.

7 TAYSIR 2023 Competition

The chapter discusses how the Neural-Checker tool was tested [15] in the TAYSIR competition.

7.1 Description of the competition

TAYSIR is an on-line competition about model inference form Neural Networks. The 2023 challenge was divided in two tracks. In this thesis we only describe the first track since the second one is out of scope. The goal of Track 1 was to learn language acceptors from Recurrent Neural Networks and Transformer classifiers. The three authors of this thesis contributed to Track 1.

7.2 Description of the tools used

We utilized Neural-Checker as the primary tool, employing version 0.38.1 of pythautomata and version 0.35.2 of pyModelExtractor. For the competition we used a representation of a DFA implemented in the tool pythautomata. Whereas the extraction algorithms used are implemented in the tool pyModelExtractor

7.3 Extraction Approach

For the extraction we used L^* algorithm for DFA with PAC since we are trying to learn ANN. For the **EQ** we need to define a distribution over sequences to sample from. We evaluated the following cases: *uniform length*, sampling from *length distribution in validation data*, and using the *full prefix set* of words up to some given length.

To guarantee termination, a maximum running time was set. This implies stopping an extraction if the run surpasses a given duration. To extract the automaton we used both the traditional and *partial* approach.

Finally, to lower the CPU time of the submitted model we used a new automata implementation, *FastDFA*, which reduced the models inference time and memory usage. However, this structure is not as general as the type of models provided by pyhtautomata, since symbols are only restricted to integer type and do not allow for more complex data structures.

7.4 Experimental Results

We present the best results regarding the competition score, however, they are mainly focused in the ER (error rate: ER $= \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[y_i \neq \hat{y}_i]$) and CT (CPU time the submitted model lasts processing a given sequence). In track 1, see Table 7.1, the datasets 2, 3, 4, 5, 6 and 7 ended up with a perfect submission ER. For these cases **EQ** was implemented with PAC, using a sampling technique that generated words with a fixed length of 22. In this setting we did not fix a maximum running time as L* execution finished with **EQ** passing the PAC test. All PAC tests were performed with $\epsilon = \delta = 0.01$ parameters. For dataset 1, 9, 10 and 11, PAC was not passed, and we resorted to use a a continuous run that stopped every 3 hours, outputted a partial DFA and then continued running with the same Observation Table. Best results were obtained on 5 to 7 iterations of this process (15-21 hours runs). For datasets 9, 10 and 11 sampling from length distribution in validation led to better results. For dataset 8 we tried all mentioned techniques but none ended with positive results. The result became more complex the longer we trained the model, making the memory usage bigger and the ER greater. We ended up submitting a trivial model that rejects (returns False) every given sequence. This turned out to be the best model.

Dataset	Duration (s)	EQs	MQs	Extracted States	Validation ER	Submission ER	CT (ms)
1	18.0	9	11.8M	38,400	0.071	0.0844	0.068
2	1.1	5	10.8k	9	0	0	0.072
3	2.0	3	3.02k	10	0	0	0.074
4	2.6	3	2.89k	5	0	0	0.057
5	1.1	2	1.16k	6	0	0	0.056
6	0.8	1	129	2	0	0.00001	0.054
7	0.3	1	129	2	0	0	0.055
8	-	-	-	1	0.342	0.327	0.030
9	1080.0	3	5.13M	33,700	0.013	0.0307	0.057
10	1080.0	1	3.09M	18,300	0.182	0.0523	0.059
11	1260.0	0	4.81M	16,000	0.249	0.0220	0.086

Table 7.1: Track 1 results

8 Conclusions

We discussed, implemented and empirically evaluated and analyzed different automata learning algorithms for black-box models, aligning with our research objectives. The intention was to create a comprehensive guide for those entering this domain, helping them to understand the principles of neural language acceptors verification and analysis through automata models.

Exploring both theory and practice made possible the comprehension of fundamental concepts, enabling us to find improvements and discover new algorithms and opportunities for optimizing existing implementations. The L^* algorithm was improved in terms of time efficiency. We evaluated the enhancements showing that the optimizations implemented have resulted in a significant improvement.

Moore and Mealy Machines have been studied and integrated into the tool, expanding the potential applications. Additionally, an extension of L^* designed for learning Moore Machines was implemented, alongside an algorithm for converting Moore Machines to minimal Mealy Machines. These enhancements increase the scope and utility of the tool.

Furthermore, we introduced the Observation Pack algorithm in the Neural-Checker tool, presenting a new discrimination tree algorithm that needs to be tested in future competitions and case studies.

Participating in TAYSIR offered us a practical driver for evaluating and proposing enhancements for the Neural-Checker tool.

We explored a wide variety of oracles and evaluated the performance of the algorithms using different types of them. We were able to gain insights into the strenghts and weaknesses of each algorithm in various oracle settings. By analyzing factors such as time efficiency and accuracy, we could evaluate the most appropriate algorithm and oracle for different scenarios.

8.1 Future Work and Open Problems

As future work and open problems, we identify two aspects: first, reduce the time execution efficiency of the currently implemented extraction algorithms; and second, improve the expressiveness of the extracted models.

When it comes to the first point, we discovered that the Observation Pack algorithm works really fast when using certain tools for black-box models. However, there is a downside, when using non-exact **EQ** methods such as PAC or Random Walk, it finishes more often than other extraction algorithms with a model that might not be exactly the same as the target one. Another issue is that the length of the queries gets longer as the teacher provides longer counterexamples, as explained in [32].

Because of these challenges, Isberner, Howar, and Steffen came up with the TTT algorithm [40]. The plan was to implement it and compare it with the Observation Pack algorithm. Unfortunately, due to the complexities associated with understanding and implementing the TTT algorithm, our original idea to directly compare it with Observation Pack was not possible. However, we still plan to improve the current implemented algorithms, which includes giving a second try to the TTT algorithm.

Turning to the second aspect, the extracted models may vary depending on the level of interpretability, verification, and complexity. Currently, Neural-Checker has models belonging to regular-grammars. These models are the least expressive. It has been demonstrated in [41, 42] that the most commonly used architectures today for LM (Language Model) implementation have a higher level of expressiveness than the models currently extracted by the team.

While there are algorithms for extracting more expressive models [43, 44, 45] at present, to the best of our knowledge, these are in their early development stages, and practical results have not been demonstrated, nor have stable tools been created for public use.

In the given context, the upcoming efforts are directed towards extracting more expressive models. This involves the initial development of efficient algorithms, followed by their subsequent implementation into the Neural-Checker tool.

Acknowledgments

Research reported in this thesis has been partially funded by ANII-Agencia Nacional de Investigación e Innovación under grant IA_1_2022_1_173516.

9 Bibliography

[1] S. Chen, Y. Sun, L. D., Q. Wang, Q. Hao, and J. Sifakis, "Runtime safety assurance for learning-enabled control of autonomous driving vehicles," *CoRR*, vol. abs/2109.13446, 2021, accessed: March 21th, 2024. [Online]. Available: https://arxiv.org/abs/2109.13446

[2] T. Mitchell, *Machine Learning*, ser. McGraw-Hill International Editions. McGraw-Hill, 1997, accessed: March 21th, 2024. [Online]. Available: https://books.google.com.uy/books?id=EoYBngEACAAJ

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org, Accessed: March 21th, 2024.

[4] F. Mayr and S. Yovine, "Regular Inference on Artificial Neural Networks," in *2nd International Cross-Domain Conference for Machine Learning and Knowledge Extraction (CD-MAKE)*, ser. Machine Learning and Knowledge Extraction, A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, Eds., vol. LNCS-11015. Hamburg, Germany: Springer International Publishing, Aug. 2018, pp. 350–369, part 5: MAKE Explainable AI, Accessed: March 21th, 2024. [Online]. Available: https://inria.hal.science/hal-02060043

[5] T. Lei, R. Barzilay, and T. Jaakkola, "Rationalizing neural predictions," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, J. Su, K. Duh, and X. Carreras, Eds. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 107–117, accessed: March 21th, 2024. [Online]. Available: https://aclanthology.org/D16-1011

[6] A. Holzinger, C. Biemann, C. S. Pattichis, and D. B. Kell, "What do we need to build explainable ai systems for the medical domain?" 2017.

[7] T. G. Calderon and J. J. Cheh, "A roadmap for future neural networks research in auditing and risk assessment," *International Journal of Accounting Information Systems*, vol. 3, no. 4, pp. 203–236, 2002, second International Research Symposium on Accounting Information Systems, Accessed: March 21th, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1467089502000684

[8] C. Zhang, J. Jiang, and M. Kamel, “Intrusion detection using hierarchical neural networks,” *Pattern Recognition Letters*, vol. 26, no. 6, pp. 779–791, 2005, accessed: March 21th, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167865504002624

[9] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, “Explaining explanations: An overview of interpretability of machine learning,” 2019.

[10] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, D. Pedreschi, and F. Giannotti, “A survey of methods for explaining black box models,” 2018.

[11] M. Grzes and M. Taylor, “Proceedings of the adaptive and learning agents workshop,” 2010.

[12] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, 2010, accessed: March 21th, 2024. [Online]. Available: https://books.google.com.tr/books?id=XAOE5V9B4dUC

[13] L. G. Valiant, “A theory of the learnable,” *Commun. ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[14] F. Mayr, S. Yovine, F. Pan, and F. Vilensky, “Neural checker,” https://github.com/orgs/neuralchecker/, 2021, accessed: March 21th, 2024.

[15] F. Mayr, S. Yovine, M. Carrasco, A. Garat, M. Iturbide, J. da Silva, and F. Vilensky, “Results of neural-checker toolbox in taysir 2023 competition,” in *Proceedings of 16th edition of the International Conference on Grammatical Inference*, ser. Proceedings of Machine Learning Research, F. Coste, F. Ouardi, and G. Rabusseau, Eds., vol. 217. PMLR, 10–13 Jul 2023, pp. 295–298, accessed: March 21th, 2024. [Online]. Available: https://proceedings.mlr.press/v217/mayr23b.html

[16] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, ser. Addison-Wesley series in computer science. Addison-Wesley, 2001, accessed: March 21th, 2024. [Online]. Available: https://books.google.com.uy/books?id=omIPAQAAMAAJ

[17] N. Chomsky, “Three models for the description of language,” *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956.

[18] J. E. Hopcroft and R. M. Karp, “A linear algorithm for testing equivalence of finite automata.” 1971, accessed: March 21th, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:120207847

[19] O. Kakde, *Theory of Computation.* Laxmi Publications Pvt Limited, 2008, accessed: March 21th, 2024. [Online]. Available: https://books.google.com.uy/books?id=y11InQEACAAJ

[20] J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. Academic Press, 1971, pp. 189–196, accessed: March 21th, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124177505500221

[21] F. Mayr Ojeda, "Regular inference over recurrent neural networks as a method for black box explainability," 2019.

[22] B. K. Aichernig, E. Muškardin, and A. Pferscher, "Active vs. passive: A comparison of automata learning paradigms for network protocols," *Electronic Proceedings in Theoretical Computer Science*, vol. 371, p. 1–19, Sep. 2022, accessed: March 21th, 2024. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.371.1

[23] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, no. 3, pp. 302–320, 1978, accessed: March 21th, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995878905624

[24] D. Angluin, "Learning regular sets from queries and counterexamples," Tech. Rep., 1987, accessed: March 21th, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540187900526?via%3Dihub

[25] E. F. Moore, *Gedanken-Experiments on Sequential Machines*, C. E. Shannon and J. McCarthy, Eds. Princeton: Princeton University Press, 1956, accessed: March 21th, 2024. [Online]. Available: https://doi.org/10.1515/9781400882618-006

[26] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[27] A. El-Maleh, "A note on moore model for sequential circuits," *ResearchGate.com. Published on July*, 2016.

[28] A. Klimowicz and V. Solov'ev, "Transformation of a mealy finite-state machine into a moore finite-state machine by splitting internal states," *Journal of Computer and Systems Sciences International*, vol. 49, pp. 900–908, 12 2010.

[29] F. Mayr, S. Yovine, M. Carrasco, F. Pan, and F. Vilensky, "A congruence-based approach to active automata learning from neural language models," in *Proceedings of 16th edition of the International Conference on Grammatical Inference*, ser. Proceedings of Machine Learning Research, F. Coste, F. Ouardi, and G. Rabusseau, Eds., vol. 217. PMLR, 10–13 Jul 2023, pp. 250–264, accessed: March 21th, 2024. [Online]. Available: https://proceedings.mlr.press/v217/mayr23a.html

[30] C. Nicaud, "Random deterministic automata," in *Mathematical Foundations of Computer Science 2014*, E. Csuhaj-Varjú, M. Dietzfelbinger, and Z. Ésik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 5–23.

[31] M. J. Kearns and U. Vazirani, *An Introduction to Computational Learning Theory.* The MIT Press, 08 1994, accessed: March 21th, 2024. [Online]. Available: https://doi.org/10.7551/mitpress/3897.001.0001

[32] M. Isberner, "Foundations of active automata learning: An algorithmic perspective," der Technischen Universität Dortmund, Germany, Tech. Rep., 2015, accessed: March 21th, 2024. [Online]. Available: https://core.ac.uk/download/pdf/46916458.pdf

[33] GeeksforGeeks. (2023) Lowest common ancestor in a binary tree — set 1. Accessed: January 14th, 2024. [Online]. Available: https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/

[34] R. Rivest and R. Schapire, "Inference of finite automata using homing sequences," Tech. Rep., 1993, accessed: March 21th, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540183710217

[35] F. M.Howar, "Active learning of interface programs," der Technischen Universität Dortmund, Germany, Tech. Rep., 2012, accessed: March 21th, 2024. [Online]. Available: https://eldorado.tu-dortmund.de/handle/2003/29486

[36] R. T. Tije, "A comparison of counterexample processing techniques in angluin-style learning algorithms," Tech. Rep., 2022, accessed: March 21th, 2024. [Online]. Available: https://www.cs.ru.nl/bachelors-theses/2022/Rick_ten_Tije___1005826___A_comparison_of_counterexample_processing_techniques_in_Angluin-style_learning_algorithms.pdf

[37] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[38] F. Xia, J. Liu, H. Nie, Y. Fu, L. Wan, and X. Kong, "Random walks: A review of algorithms and applications," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 4, no. 2, p. 95–107, Apr. 2020, accessed: March 21th, 2024. [Online]. Available: http://dx.doi.org/10.1109/TETCI.2019.2952908

[39] E. Muškardin, B. K. Aichernig, I. Pill, A. Pferscher, and M. Tappler, "Aalpy: An active automata learning library," in *Automated Technology for Verification and Analysis*, Z. Hou and V. Ganesh, Eds. Cham: Springer International Publishing, 2021, pp. 67–73.

[40] M. Isberner, F. Howar, and B. Steffen, "The ttt algorithm: A redundancy-free approach to active automata learning," in *Runtime Verification*, 2014, accessed: March 21th, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:15853481

[41] J. Pérez, J. Marinkovic, and P. Barceló, "On the turing completeness of modern neural network architectures," *CoRR*, vol. abs/1901.03429, 2019, accessed: March 21th, 2024. [Online]. Available: http://arxiv.org/abs/1901.03429

[42] H. Siegelmann and E. Sontag, "On the computational power of neural nets," *Journal of Computer and System Sciences*, vol. 50, no. 1, pp. 132–150, 1995, accessed: March 21th, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000085710136

[43] J. Michaliszyn and J. Otop, "Learning Deterministic Visibly Pushdown Automata Under Accessible Stack," in *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Szeider, R. Ganian, and A. Silva, Eds., vol. 241. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 74:1–74:16, accessed: March 21th, 2024. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.MFCS.2022.74

[44] B. Barbot, B. Bollig, A. Finkel, S. Haddad, I. Khmelnitsky, M. Leucker, D. Neider, R. Roy, and L. Ye, "Extracting context-free grammars from recurrent neural networks using tree-automata learning and a* search," in *Proceedings of the Fifteenth International Conference on Grammatical Inference*, ser. Proceedings of Machine Learning Research, J. Chandlee, R. Eyraud, J. Heinz, A. Jardine, and M. van Zaanen, Eds., vol. 153. PMLR, 23–27 Aug 2021, pp. 113–129, accessed: March 21th, 2024. [Online]. Available: https://proceedings.mlr.press/v153/barbot21a.html

[45] D. M. Yellin and G. Weiss, “Synthesizing context-free grammars from recurrent neural networks (extended version),” *CoRR*, vol. abs/2101.08200, 2021, accessed: March 21th, 2024. [Online]. Available: https://arxiv.org/abs/2101.08200