Universidad ORT Uruguay
Facultad de Ingeniería

Active Learning Over Large Alphabets

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

Federico Vilensky - 185975

Tutores: Franz Mayr y Sergio Yovine

2022

Declaración de Autoría

Yo, Federico Vilensky declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

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

Federico Vilensky

28-09-2022

Agradecimientos

Dedicado a mi familia y amigos, quienes fueron mi soporte en esta etapa.

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

Abstract Español

El presente trabajo es una extención y reestructura del framework de algoritmos de aprendizaje de autómatas basados en MAT, desarrollado por Mayr y Yovine en Universidad ORT. El objetivo del trabajo es doble, primero re-arquitecturar el framework para que sea más fácilmente extensible, y segundo, agregar un algoritmo de aprendizaje de autómatas simbólicos. Se decidió el uso de autómatas simbólicos por la facilidad que pueden tener para la representación de alfabetos grandes, potencialmente infinitos. Una vez implementado, se comparará el algoritmo contra el algoritmo desarrollado por Angluin.

Abstract

This work is an extension and rework of the automata MAT learning framework developed by Mayr and Yovine at Universidad ORT. The goal is twofold, to re-architecture the framework so that it's more easily extensible, and to add and evaluate an algorithm for learning symbolic automata. Symbolic automata were chosen because of the possibility of learning large, possibly infinite, alphabets. Once the algorithm is implemented, we will compare the performance compared to Angluin's algorithm.

Palabras clave

Clasificación de secuencias; Autómata finito determinístico; Autómata finito simbólico; Extracción de reglas; Inferencia regular; Inteligencia Artificial; Explicabilidad; Alfabetos grandes; L*

Key words

Sequence classification; Deterministic finite automata; Symbolic finite automata; Rule extraction; Regular inference; Artificial Intelligence; Explainability; Large Alphabets; L*

Contents

1 Introduction

We are interested in the general problem of learning finite state automata from a black-box system with large alphabets. The task of constructing an automaton that behaves as the black box is called *identification* or *regular inference* [1].

This task can be carried out in the active or passive learning paradigms. In active learning, the learner interacts with the black box at training time, while the passive learner only observes the information provided without influencing or directing it [2].

In this work, we will focus on the active paradigm, following adaptations of L^*[3] that work on large and possibly infinite alphabets. For this we will resort to Drew's and D'antoni's [4] adaptation.

This adaptation, named Λ^* falls in the Minimally Adequate Teacher category (MAT) proposed by Angluin[3], this means Λ^* actively learns an automata inside a black-box by interacting through two types of queries: Membership Queries (**MQ**) and Equivalence Queries (**EQ**).

On top of that kind of interactions, Λ^* relies in Boolean algebra learners, that allow for the construction of syntactically succinct representations of transitions that would otherwise be represented in an extensive manner.

The objective of this work is to give a step in the direction of actively learning finite state automata with large alphabets. For that, we implemented and adapted Λ^* following good design and architectural patterns. This implied the re-architecture of legacy code, development, testing and evaluation of the aforementioned technique.

Outline

In Chapter 2 we present the necessary previous knowledge and an analysis of the problem in question.

In Chapter 3 we discuss different approaches to this problem, and present our version of the solution to it.

In Chapter 4 we show and explain the architecture of the new framework.

In Chapter 5 we test Λ^* performance against L^* and against itself with different Boolean algebras.

2 Preliminaries

2.1 Learning Regular Languages

2.1.1 Regular Languages

Regular languages can be defined as the ones that can be described by deterministic finite automata (DFA) [5]. DFA are formally defined as a tuple $(Q,\ \Sigma,\ \delta,\ q_0,\ F)$ where:

1. Q is a finite set of states.
2. Σ is a finite set of input symbols.
3. δ is a transition function that takes as arguments a state and an input symbol and returns a state. $\delta : Q \times \Sigma \to Q$
4. q_0 is a start state, belonging to Q.
5. F is a set of final or accepting states, F being a subset of Q.

The Chomsky hierarchy defines these languages as the languages that are generated by Type-3 grammars (regular grammars) [6].

From the description in [5], the key point is that these models characterise the languages, meaning they provide a constructive way of describing them, and recognising their elements (that is, checking whether a sequence of symbols does belong to the language).

A simple example of a regular language is presented in Figure 2.1. This language is described by the regular expression $(ab)^*$, with:

1. $Q = \{0, 1, 2\}$.

2. $\Sigma = \{a, b\}$ and λ being the empty sequence.

3. δ as presented graphically in the figure or tabularly in Table 2.1.

4. $q_0 = 0$ (indicated with an incoming arrow).

5. $F = \{0\}$ (indicated with a double circle).

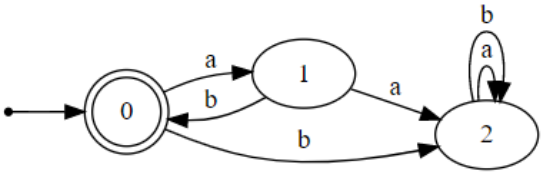

Figure 2.1: Example of automaton

δ	**a**	**b**
0	1	2
1	2	0
2	2	2

Table 2.1: Table of transition function δ of automaton in Fig. 2.1

2.1.2 Grammatical Inference

Grammatical Inference is defined as the problem of inducing, learning or inferring grammars. It is a field with connections to a series of disciplines such as bio-informatics, computational linguistics and pattern recognition [1]. The goal of this field is to infer grammars given some information about the languages, and, as grammars are constructive models, they present insight and generalisation over the words belonging to the language, something that most state of the art neural models lack.

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

Passive learning consists in learning a language from a set of given positive and/or negative examples [7]. It has been shown in [8] that finding a minimal DFA that is consistent with a given arbitrary set of sequences is *NP-complete*.

In the active learning setting, the learner is given the ability to draw examples and to ask membership queries to the teacher. A well known algorithm in this category is Angluin's L^* [3]. L^* is *polynomial* in the number of states of the minimal deterministic finite automaton (DFA) and the maximum length of any sequence exhibited by the teacher.

2.1.3 L^*

L^* constructs a DFA by interacting with a Minimal 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 test* (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 an arbitrary counterexample (a word belonging to one of the languages but not the other).

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 through the Membership Oracle (MQ) 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 by two sets of rows: the 'upper' rows (or top part, that we will call **RED** following De la Higuera's notation [1]), that represent the elements of the prefix-closed set of strings mentioned earlier, and the 'lower' rows (or bottom part, that we will call **BLUE**), which represent the same elements of this set but concatenated with the set of letters 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 2.2b.

The observation table is first initialized by building one **RED** row (for the empty word λ) and one **BLUE** row for each symbol in the alphabet Σ (length-one words). Then the iterative process begins.

In order to be able 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 the bottom part of the table, there is an equal row in the top part. The second property is consistency. A table is considered consistent if for every pair of rows in the top part of the table (**RED**) with the same values (same order of 0s and 1s), then all pairs of extensions with the same letter of the alphabet must have the same row in the table. Precisely, a table is consistent if for every different row in **RED**, for every symbol x in Σ if $OT[v] = OT[w]$ then $OT[v.x] = OT[w.x]$.

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

To make it consistent, the algorithm expands the original set of suffixes with the letter that makes their corresponding extensions different (an $x \in \Sigma$ such that $OT[v] = OT[w]$ but $OT[v.x] \neq OT[w.x]$). This is done in order to differentiate between the two words that had the same row values.

Once the table is closed and consistent, the algorithm proceeds to construct the conjectured DFA and then asks the oracle 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 counter example. 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).

2.1.3.1 An L^* run

Let us take as an example the regular language presented in Figure 2.1, described by the regular expression $(ab)^*$.

First, the algorithm constructs the table as presented in 2.2a. 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, one of the elements in **BLUE** that has not a representative in **RED** is selected, for example a, and moves it to **RED**, adding to **BLUE** its concatenation to every symbol (aa and ab, then the table is filled by asking the corresponding **MQ**s). The resulting table can be seen in Table 2.2b.

As the observation table is now closed (the previous step solved this problem) and consistent an automaton can be built.

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 $OT[w][\lambda] = 1$, and rejecting states are those rows v where $OT[v][\lambda] = 0$. Finally the transition function is defined as: $\delta(q_v, a) = q_w$ if $OT[va] = OT[w]$. The resulting automaton is presented in Figure 2.2.

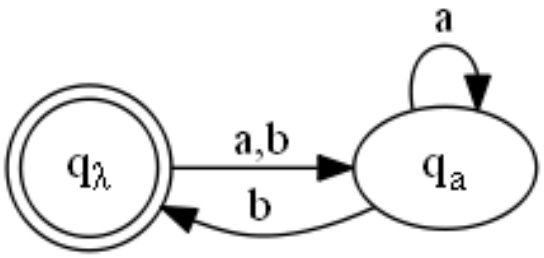

Figure 2.2: First proposed automaton in an L^*example run.

This automaton is then presented to the teacher via the **EQ**, which can be implemented by the table-filling algorithm [5]. 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 '*bb*'.

Now, the learner proceeds to process the counterexample. This is done by adding the counterexample and all its prefixes to **RED**, and at the same time adding for each prefix v and for all symbol x, $v.x$ to **BLUE**, given that $v.x$ is not a prefix of the counterexample. Then holes are filled, resulting in the Table 2.2c

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, $OT[a] = OT[b]$, however $OT[ab] \neq OT[bb]$). 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 to the columns of the observation table (in this case symbol b). The symbol is added, holes are filled, the result is Table 2.2d.

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

OT_0	λ
λ	1
a	0
b	0

(a)

OT_1	λ
λ	1
a	0
b	0
aa	0
ab	1

(b)

OT_2	λ
λ	1
a	0
b	0
bb	0
aa	0
ab	1
ba	0
bba	0
bbb	0

(c)

OT_3	λ	b
λ	1	0
a	0	1
b	0	0
bb	0	0
aa	0	0
ab	1	0
ba	0	0
bba	0	0
bbb	0	0

(d)

Table 2.2: Observation tables during an L^* example run.

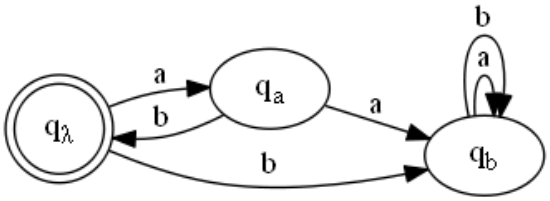

Figure 2.3: Output automaton in an L^* example run.

2.1.4 Boolean algebra

In symbolic automata, transitions, instead of being represented by a symbol, are represented by predicates over a decidable Boolean algebra. A Boolean Algebra $\mathcal{A}$ is a tuple $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is the set that represents the domain; Ψ is a set of predicates closed under Boolean connectives ($\vee$, $\wedge$, and $\neg$), with $\bot, \top \in \Psi$; $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ is a function defined as such (i) $[\![\bot]\!] = \{\}$ (ii) $[\![\top]\!] = \mathfrak{D}$, and (iii) $\forall \phi, \psi \in \Psi, [\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$, $[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!]$, and $[\![\neg\phi]\!] = \mathfrak{D} \setminus [\![\phi]\!]$ For example:

- Equality Algebra: The equality algebra for any set $\mathfrak{D}$ has predicates formed from combinations of formulas with the form $\lambda x.x = a$, denoted ψ_a for any $a \in \mathfrak{D}$. Formally we would say that Ψ is the Boolean closure of $\Psi_0 = \{\psi_a | a \in \mathfrak{D}\} \cup \{\bot, \top\}$, where $\forall a \in \mathfrak{D}, [\![\psi_a]\!] = \{a\}$. Some predicates we can form with this algebra are $\lambda x.\neg x = 2$ and $\lambda x.\neg x = 5 \wedge \neg x = 10$.

- Closed Interval Algebra: The union of left-closed right-closed intervals over integers (i.e. $\mathbb{Z}$). Take the Boolean closure of $\Psi_0 = \{\psi_{i,j} | i, j \in \mathbb{Z} \wedge i < j\} \cup \{\bot, \top\}$ where $[\![\psi_{i,j}]\!] = [i, j]$. Some predicates we can form with this algebra (using mathematical interval notation) are $[0, 10] \cup [42, 100]$ and $(-\infty, -10] \cup [10, +\infty)$

2.1.5 Boolean Algebra Learner

A Boolean algebra learner is a helper function we use to generate predicates from an existing DFA. For every pair of states (q_s,q_t) the algebra learner will create a single predicate in a given Boolean algebra that will represent every single transition from q_s to q_t in the DFA, and return the transitions corresponding to an equivalent SFA. If we then substitute the DFA's transitions for these predicates, we get an SFA. For example, given the transitions from the DFA in Figure 2.4:
$\delta = \{(q_{init}, a, q_1), (q_{init}, b, q_1), (q_{init}, c, q_1), (q_{init}, d, q_{init}), (q_{init}, e, q_{init}), (q_1, a, q_{init}),$
$(q_1, b, q_{init}), (q_1, d, q_{init}), (q_1, c, q_1), (q_1, e, q_1)\}$

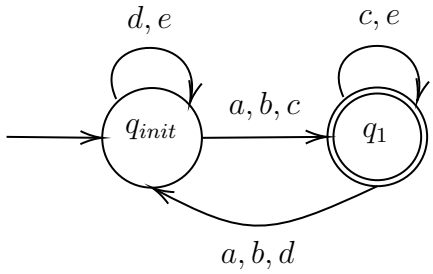

Figure 2.4: DFA

An equality algebra learner that received the transitions from Figure 2.4, would result in the transitions corresponding to Figure 2.5:
$\Delta = \{(q_{init}, \lambda x.(x = a) \vee (x = b) \vee (x = c), q_1), (q_{init}, \lambda x.(x = d) \vee (x = e), q_{init}), (q_1, \lambda x.(x = a) \vee (x = b) \vee (x = d), q_{init}), (q_1, \lambda x.(x = c) \vee (x = e), q_1)\}$

A closed interval algebra that received the transitions from Figure 2.4, would result in the transitions corresponding to Figure 2.6, assuming the characters are ordered:
$\Delta = \{(q_{init}, [a, c], q_1), (q_{init}, [d, e], q_{init}), (q_1, [a, b] \cup [d, d], q_{init}), (q_1, [c, c] \cup [e, e], q_1)\}$

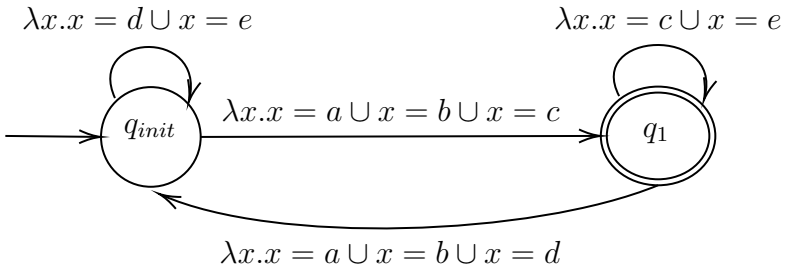

Figure 2.5: Equality Boolean Algebra SFA

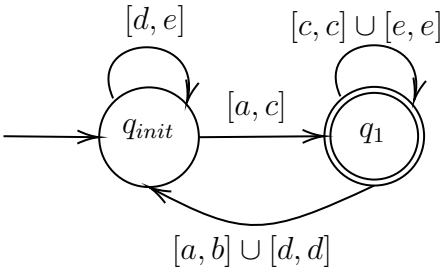

Figure 2.6: Closed Intervals Boolean Algebra SFA

2.1.6 Symbolic Finite Automata

A symbolic finite automaton (SFA) is formally defined as a tuple $(\mathcal{A}, Q, \Delta, q_0, F)$ where

1. $\mathcal{A}$ is an effective Boolean algebra, called the alphabet.
2. Q is a finite set of states.
3. q_0 is the initial state, $q_0 \in Q$.
4. F is set of final or accepting states, $F \subseteq Q$.
5. $\Delta \subseteq Q \times \Psi \times Q$ is the transition relation, consisting of transitions from the source state to the target state, also denoted as $q_s \xrightarrow{\psi} q_t$.

Characters are elements of $\mathfrak{D}_{\mathcal{A}}$, words are either empty (denoted by λ) or a finite sequence of characters.

A transition $\rho = (q_s, \psi, q_t) \in \Delta$ also denoted $q_s \xrightarrow{\psi} q_t$ is a transition from q_s, the source state, to q_t, the target state, where ψ is the guard or predicate of the transition. For a transition to be feasible, the predicate must be satisfiable.

For a character $a \in \mathfrak{D}_{\mathcal{A}}$ in a given automaton, an a-move, denoted $q_s \xrightarrow{a} q_t$ is a transition $q_s \xrightarrow{\psi} q_t / a \in [\![\psi]\!]$. An SFA is *deterministic* if $\forall (q_1, \psi_1, q_2)(q_1, \psi_2, q_2) \in \Delta, q_1 \neq q_2 \Rightarrow [\![\psi_1 \wedge \psi_2]\!] = \{\}$, i.e. for each starting state q_s and character a there is at most one a-move. An SFA is complete if, $\forall q \in Q, \bigvee_{(q_s, \psi_i, q_t) \in \Delta} \psi_i \ = \top$, i.e. for each state q_s and character a, there exists an a-move out of q_s. From here on, we will assume every SFA is both deterministic and complete, we can make this assumption because every SFA can be determinized and completed [9].

3 Learning Over Large Alphabets

We address the following problem: given a language defined over a *large* alphabet Σ, how can we actively learn it? This *large* alphabet is a language that is either infinite, like $\mathbb{N}$ or $\mathbb{Z}$, or simply large enough so that it's not viable to do an L^* run, as studied by Mens and Maler[10], and Drews and D'Antoni[4].

The problem arises because in L^* we need to enumerate every symbol of the alphabet to build our observation table. This will end up taking too much time, or never finish if Σ is infinite, since at each iteration, it must do a membership query to the oracle for each character of Σ. The approach Mens and Maler took is to build a table that represents the symbolic state, then build the SFA from the table. On the other hand, Drews and D'Antoni's approach is to build an incomplete deterministic finite automaton, and from there build an SFA. We will face the problem following Drews and D'Antoni's approach[4]. We will use a symbolic representation, where transitions are predicates applicable to elements of Σ, and instead of querying for each character of Σ, we will do a query for each character of a subset of Σ.

3.1 Λ^*

To solve this problem, as stated previously, we will use the Λ^* algorithm. Λ^* builds on top of L^*. So like L^*, the algorithm tries to learn an automaton, called the *target*, interacting with it only via an oracle that provides us with two queries. A *membership query* (**MQ**) and an *equivalence query* (**EQ**). Λ^*, like L^*, uses an observation table, but unlike L^*, Λ^* does not build the solution from it, but an intermediary solution, called *evidence automaton*. This *evidence automaton* is a finite automaton that is deterministic, but not complete, for simplicity it will be referred to as a DFA, but let us not forget that it needs not be complete. Using this *evidence automaton* and a *Boolean algebra learner* we will build a deterministic

and complete SFA. One assumption that is made by Drews and D'Antoni [4] is that the Λ^* learner should know Σ, we will show later that we do not need to know Σ if it is discrete, e.g. $\Sigma = \mathbb{N}$ or $\Sigma = \mathbb{Z}$, we just need an appropriate *Boolean algebra learner*.

3.1.1 Algorithm

As we can see in Figure 3.1, it does not seem too dissimilar to L^*, but with an in-between step between doing the **MQ**s and the **EQ**.

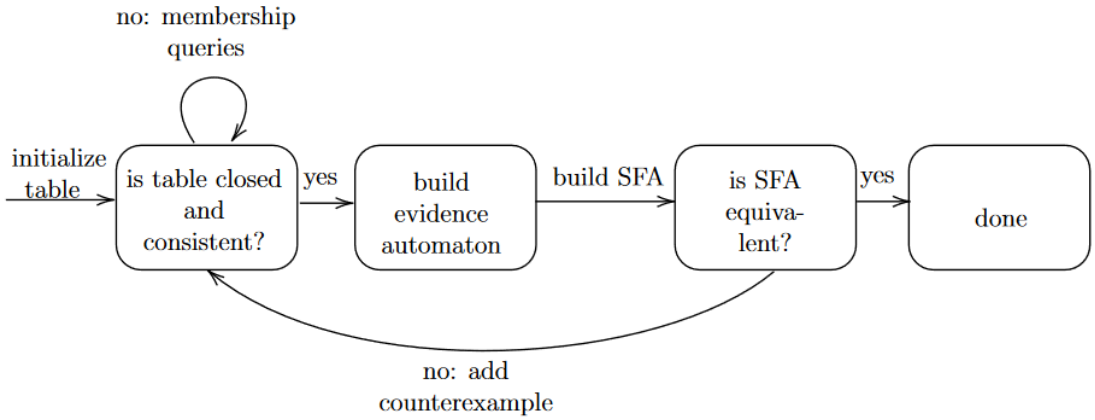

Figure 3.1: Overview of the Λ^* algorithm

Since as we said before, in this version, we do not know Σ, we cannot initialise the observation table with an arbitrary character, so what we do in this version of Λ^* is we create a model with only one state, and ask for a counterexample. This will give us at least one character to start our algorithm.

Algorithm 1: Λ^*

Input : $\mathcal{O}$:MQ, $\mathcal{E}$:EQ, Λ: Algebra learner
Output: SFA

```
σ ← ∅;
OT ← InitializeObservationTable();
H ← BuildSfa(Λ);
CounterExample, Answer ← E(H);
OT, σ ← UpdateWithNewCounterExample(OT, σ, O,CounterExample);
if OT is not consistent then
    OT,← MakeConsistent(OT, O);
end
repeat
    H ← BuildSfa(Λ);
    CounterExample, Answer ← E(H);
    if Not Answer then
        OT, σ ← UpdateWithNewCounterExample(OT, σ,
         O,CounterExample);
        if OT is not closed then
            OT,← CloseTable(OT, O);
        end
        if OT is not consistent then
            OT,← MakeConsistent(OT, σ, O);
        end
    end
until Answer = Yes;
return H
```

The set σ represents the subset of Σ, which contains all observed characters so far.

In the function *UpdateWithNewCounterExample*, we update σ with any new characters received in the counterexample, and if there are any, we also concatenate every row on our **RED** section of the observation table with these new characters.

Every time we would use Σ in L^*, we substitute it for σ, so instead of needing to enumerate every character of the alphabet, we just enumerate every character in the observed symbols set.

Otherwise this algorithm works similar to L^*.

Something to remark, is that since we are now comparing a DFA with an SFA, the equivalence oracle must be able to compare both types of automaton.

In this case, this meant adapting Hopcroft-Karp comparison algorithm to accept a symbolic automaton. The way it works is to ask both automata what their alphabets are, the DFA will answer Σ, while the SFA will answer σ, we then join both alphabets into $\Sigma' = \Sigma \cup \sigma$. This means that an automaton might need to process an unknown symbol. So every automaton must know what to do when processing an unknown symbol. In the case of SFA, this might not be a problem, but in the case of a DFA, this will be a problem. Luckily, this can be solved by adding a hole state to the automata so that any unknown symbol will end up in this state (we will add this hole to every automata for simplicity).

4 Tool Development

The Universidad ORT's AI research group already had an existing prototypical code base containing DFA, other automata variants, some automata learning algorithms like L^* and other extensions, but it was not workable. This is because it was built ad-hoc, with insurmountable amounts of technical debt, due to being written simply to experiment. This had a major consequence, it was too hard to understand what the code was doing. So it was virtually unusable by any other third party, since a third party would not have access to the team that wrote the code. This is why the decision to rewrite the whole framework was taken. Before the change, everything was written in the same library, and there was no clearly defined module structure, other than the experiments were written on a separate folder.

It was decided to split the code base into separate libraries, two published libraries for both automata and learning algorithms, and a third unpublished one for experiments as we can see in 4.1

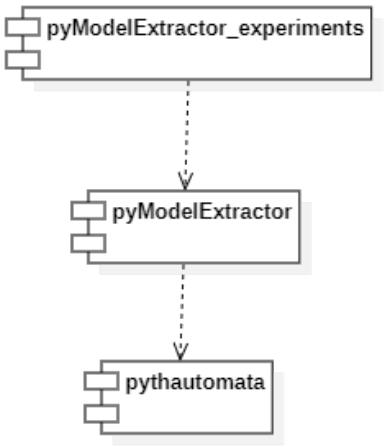

Figure 4.1: Component diagram

It was split like this so that in the case automata are needed independently from the learning algorithms, they need not be re-implemented, nor unneeded code will be imported. On top of that, being split into different projects, it does not allow couple the automata definitions to the learning algorithm. This allows us to follow a leveled architecture, which will be evident when we study the package diagrams 4.3 and 4.5.

As stated, the code was split into two separate projects. The first one, named ***pythautomata***, contains the different automata definitions. The second one, named ***pyModelExtractor***, contains the different learning algorithms. The latter also has some MAT defined (including an automaton teacher, and PAC teachers). There is an extra project, but this one is not published anywhere, while the code is still public on github. This extra project contains experiments, since they are to be used once and thrown away, they do not have to comply to the same code standards as the rest of the code base, ***poetry*** is still used inside this experiments package for dependency management.

One big improvement over the existing code is that we are publishing the new framework on PyPI, under the names ***pythautomata*** and ***pyModelExtractor***. Which means it can now be shared among other teams outside of ORT's research team. Publishing is done using the ***poetry*** tool, and github actions.

Poetry's usage is twofold. Firstly it's used to manage dependencies, this is a great advantage since it declares the project's dependencies directly inside the project, unlike vanilla python in which they are installed globally. Secondly, it

offers a tool to easily publish the project in PyPI, without any need to manually create the wheel, then logging in to PyPI and uploading the wheel. This is all handled by ***poetry*** with a single command.

The second key tool is ***Github Actions***. This tool allows the automation of either predefined or user defined actions (an action is anything you can do via a shell, usually bash) on a trigger defined by the user. In these projects, a "publish action" was created. This action runs whenever a release is created, it then bumps the project version to the version defined in the release tag, and then publishes the release in PyPI. It can then be accessed via pip, or on the PyPI web repository (e.g.: `https://pypi.org/project/pythautomata/`)

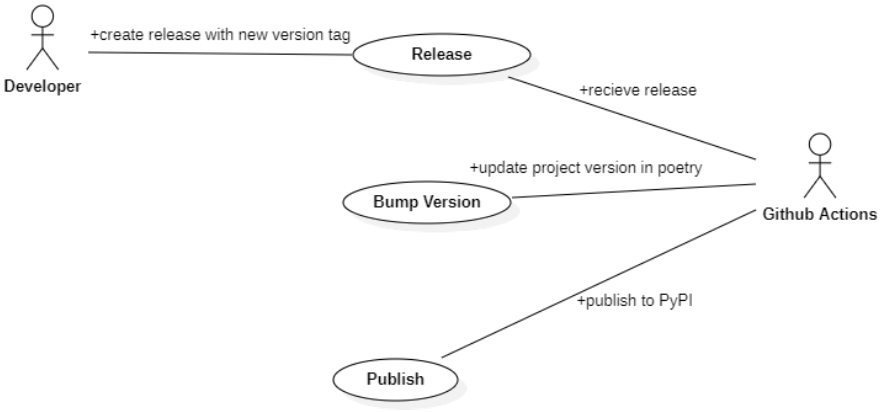

Figure 4.2: Release and publish use case

As we can see in the Figure 4.2, the github action workflow is straight forward, once a new release is published with a new version tag, the workflow will parse the release to get the new version number and then use the command
`\textit{\textbf{poetry}} version <new-version-number>`
to bump the project version to the latest release's version. It will then publish the project to PyPI using the following command
`\textit{\textbf{poetry}} publish --build -u <username> -p <pypi-api-key>`.
The ***poetry*** tool will then build the project in the correct way expected by PyPI and subsequently publish it to said platform.

On top of all this, there now exists a website hosted on *github* containing the documentation of the ***pythautomata*** package, so you can look at it in a

friendly way. One can find the page in the following link `https://neuralchecker.github.io/pythautomata/`. This documentation is generated from the *doc strings* in the pythautomata code base with the tool ***sphinx***[1].

4.1 Packages

4.1.1 pythautomata

Inside the pythautomata project there are various packages defined, split by responsibilities as we can see in 4.3.

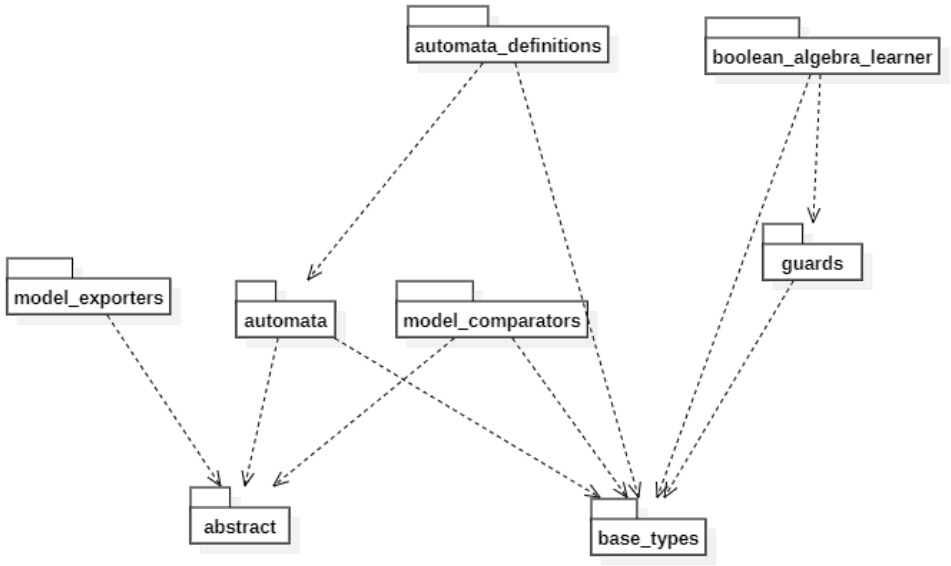

Figure 4.3: pythautomata package diagram

The first one that should be mentioned is *base_types*. In this package, as the name suggests, base types are defined. This base types are the basic language theory concepts. These includes the concept of alphabet, sequence, symbol, and variants. Regarding symbolic automata, the concept of guard and symbolic state

[1]sphinx: https://www.sphinx-doc.org/en/master/

are defined in this package. The state, has transitions to other states, the transitions are a dictionary that has a symbol as a key and a state as a value. However, in the case of the symbolic state, transitions are defined as a dictionary that has a guard as a key and a state as a value.

Then the *abstract* package should be talked about. The main thing to be mentioned is the *FiniteAutomaton* abstract class. This class provides a common interface to be used by every type of finite automaton, such as DFA, NFA, or SFA.

Thirdly, there is the *automata* package. The different automata types (i.e. DFA, SFA, etc.) are defined here. These automata contain an initial state and a set of states. Inside the same package, there are already two different algebra learners defined, which can be used or viewed as an example if a new one is to be created. These two boolean algebra learners are an equality learner, and a closed discrete intervals learner. The closed discrete intervals learner is noteworthy, it works on any alphabet of ordinal symbols. That is any set of symbols that can have the concept of "bigger than" and "smaller than". It will create a set of disjoint guards containing the appropriate intervals, having the union of the intervals be $(-\infty, \infty)$.

Speaking of automata, there is the *automata_definitions* package that has some classic example automata defined inside of it. For example Tomita's automata [11] are defined inside this package.

Then the package *guards* should be mentioned. This package is where specific guard definitions are located. For example the *ClosedIntervalGuard* and *EqualityGuard*, these being a guard that will return true to any symbol contained bewteen the limits defined in the guard, and a guard that will return true if the symbol is the same as the one defined in the guard, respectively. But more importantly, some more guards that allow combining other guards are located in this package. These are the

- *AndGuard* and *OrGuard*, that allow joining two guards with the $\wedge$ and $\vee$ operators respectively.
- *NegationGuard*, that allows to negate an existing guard with the $\neg$ operator.
- *UnionGuard* and *IntersecionGuard*, that allow joining any number of guards with the $\cup$ and $\cap$ operators respectively.

The second to last noteworthy package is *model_comparators*. This package contains comparison strategies, including the previously mentioned Hopkroft-Karp comparison algorithm.

The last package that will be mentioned is the *tests* package. As the name suggests, this package contains tests. This has two major benefits, first and foremost, it allows us to verify the API is defined correctly and behaves correctly (at least on those cases defined in the test cases). Secondly, the tests cases work as a sort of documentation, in the sense that they are usually small snippets of code that are easy enough to read, and we can easily understand what the snippets are doing. So that if the published documentation is not enough, one can always look at the tests and find out what they do.

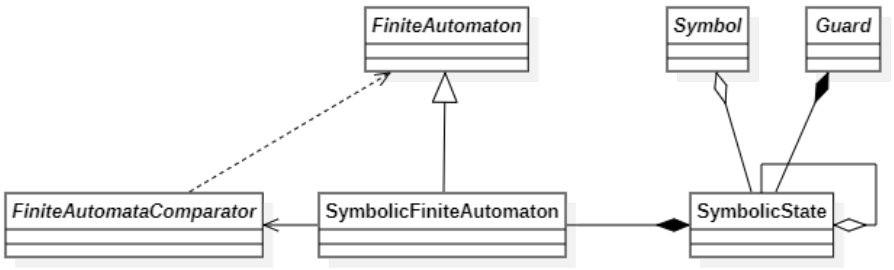

Figure 4.4: SymbolicAutomaton class diagram

As Figure 4.4 shows, the *SymbolicAutomata* will have an *AutomataComparator* and will be formed by many *SymbolicState*'s, each of them formed by pairs of guards and another state.

4.1.2 pyModelExtractor

This project has a far more simple package structure as we can see in Figure 4.5.

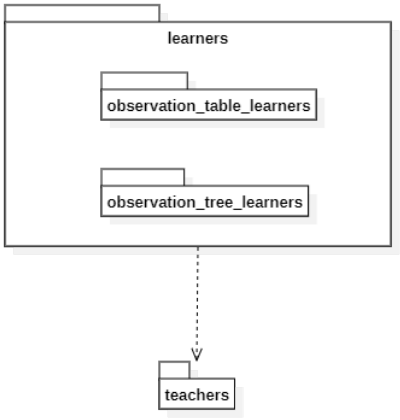

Figure 4.5: pyModelExtractor package diagram

Firstly there is the ***teachers*** package. This package contains the definition for the MAT interface to be fulfilled by the different teachers. These include the automaton teacher which is what will be used in here. Other definitions include different PAC teachers among others. The teachers must implement both the membership query and the equivalence query.

Then there is the ***learners*** package. Inside this package there is the abstract class *Learner* which defines the interface all learners must comply with. Besides this, there are two other packages ***observation_table_learners*** and ***observation_tree_learners***, each having a family of learning methods, one having those ones that use an observation table, and another having those ones that use an observation tree/trie. Inside the *observation_table_learners* is the *LambdaStarLearner* which is the class that implements the Λ^* algorithm.

Lastly, like with the previous project, there is also a *tests* package. This package serves both the same purposes. Firstly it allows one to verify the code is working correctly on the tested cases, and it also works as a use case documentation.

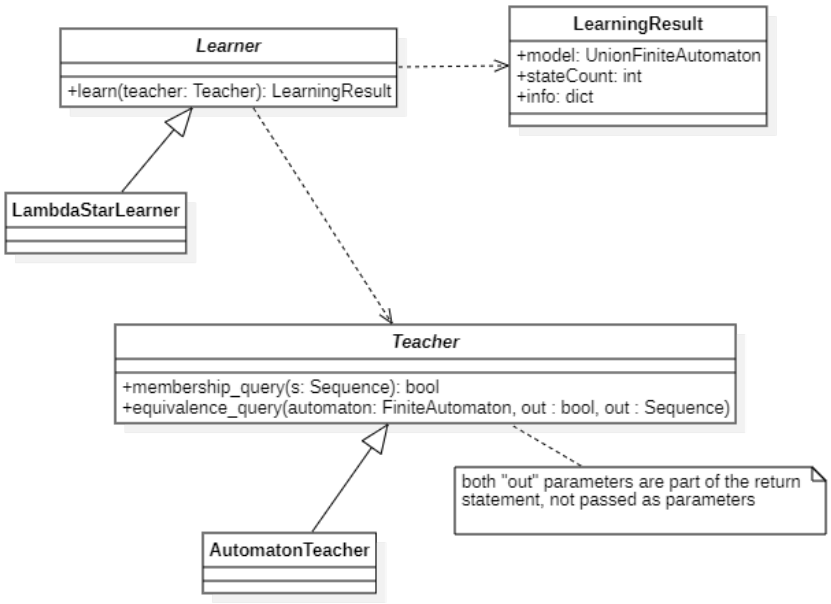

Figure 4.6: LambdaStarLearner class diagram

The Figure 4.6 shows how the *LambdaStarLearner* will communicate with the *AutomatonTeacher*. The method *learn* will receive an object with type *Teacher* and use the *equivalence_query* method (the **EQ**) and the *membership_query* method (the **MQ**) as described in the algorithm in Figure 3.

5 Experimental results

L^* algorithm will be compared against Λ^* algorithm in several tasks.

5.1 Example 1

In this first experiment, Tomita's automata [11] will be used to compare L^* against Λ^*. Tomita defined a series of regular grammars, which are used as a rite of passage for any new algorithm, so this will be the first test to compare L^* and Λ^* with a closed interval Boolean algebra learner. We will run both algorithms 10000 times with each automaton. If we take a look at Figure 5.1 we can see the DFA that represents the first grammar defined by Tomita, and if we take a look at Figure 5.2 we can see an equivalent SFA with a closed interval Boolean algebra.

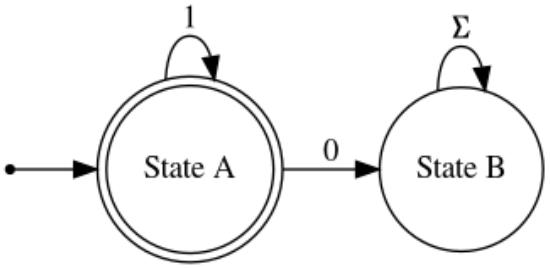

Figure 5.1: Tomita's 1^{st} DFA representation

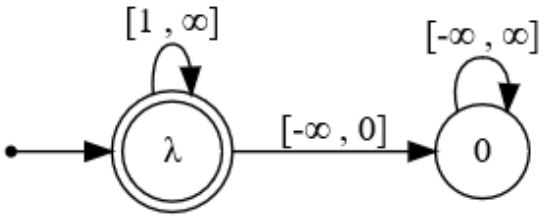

Figure 5.2: Tomita's 1^{st} SFA representation

We will be looking at **MQ**, **EQ** and time taken to learn the automata. By looking at the algorithm, Λ^* should be slower, since it does not know the full alphabet, and it needs to build first the evidence automaton and later the SFA which can be a costly operation. So without further ado, let us see the results.

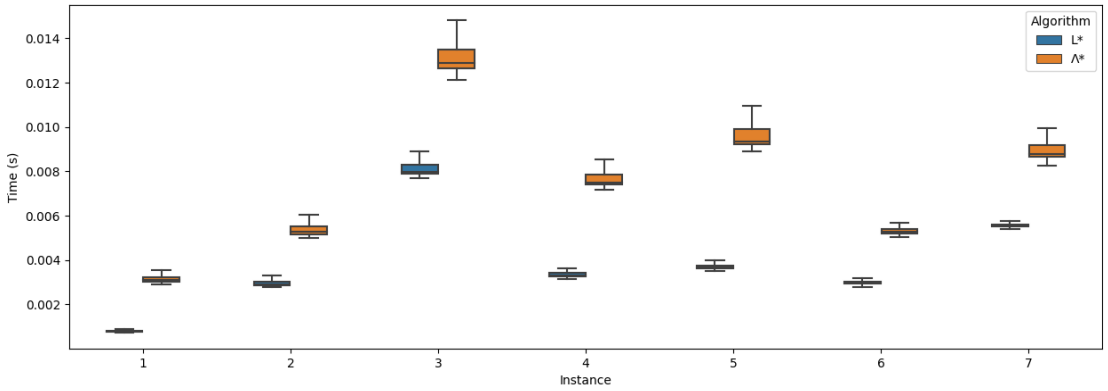

Figure 5.3: Time box plot

The box plot in 5.3 shows the time it took to run the algorithm 10000 times with each automata. The label on the "Instance" axis represents the algorithm ran and against which Tomita's grammar it was ran. For example "L^* 1" means L^* against the first grammar, whereas "Λ^* 4" means Λ^* against the fourth grammar.

As predicted, Λ^* does indeed take significantly longer to run. we will proceed to look at the **MQ** and **EQ** count to see if this was of impact.

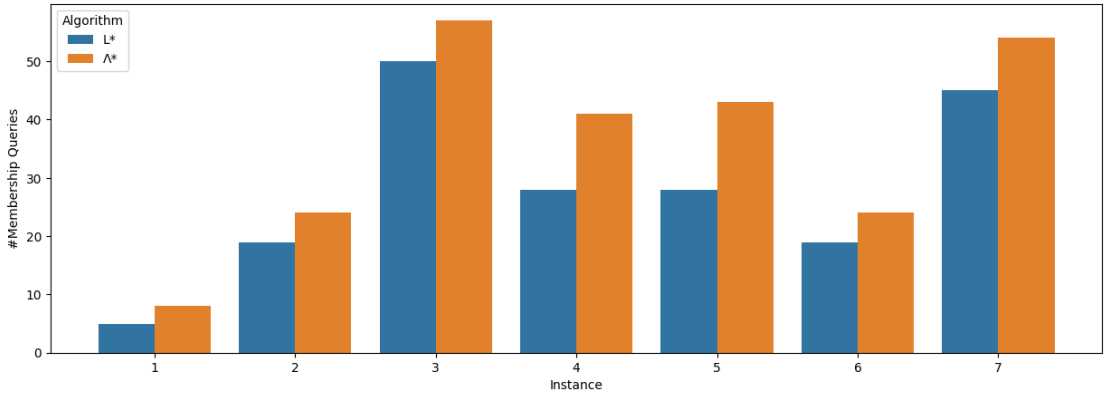

Figure 5.4: **MQ** count

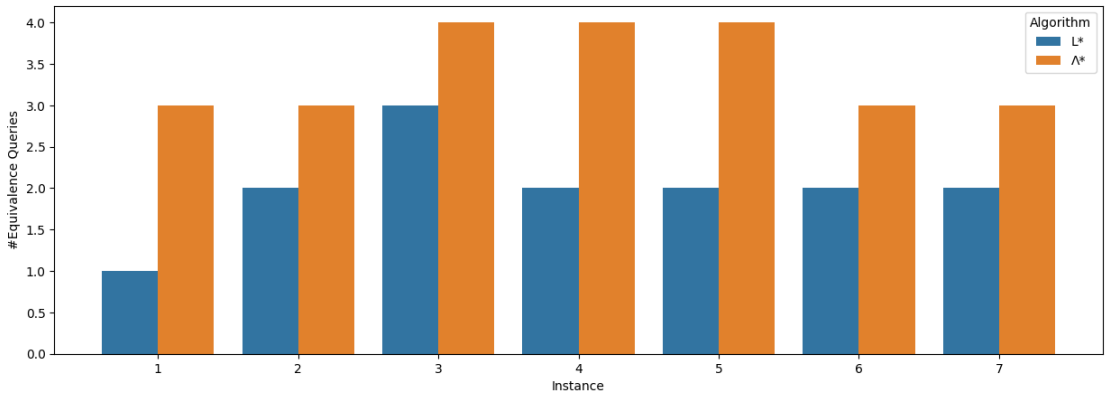

Figure 5.5: **EQ** count

We can see clearly that Λ^* indeed does need more **MQ** and **EQ**. Since **MQ** are rather fast, we will assume for now, that the difference in time was from the number of **EQ** and from the Boolean algebra learner. So what we will do next is evaluate it with a simpler Boolean algebra learner.

5.2 Example 2

We will run the same experiment as before, but this time we will run Λ^* with an equality Boolean algebra learner and compare it to the previous Λ^* run. So the resulting automaton would result in guards formed by the union of elements, for example we can compare Tomita's third grammar, the DFA defined in Figure 5.6 and the SFA defined in Figure 5.7.

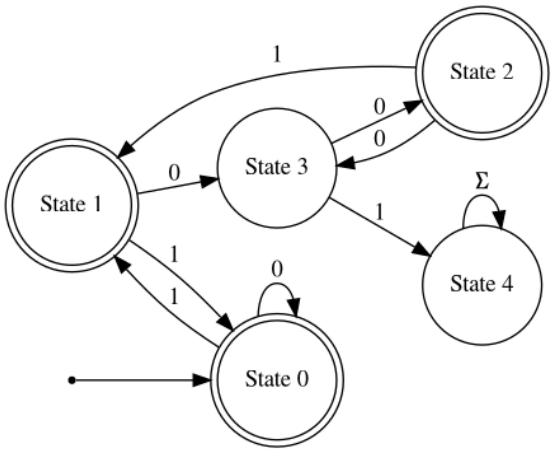

Figure 5.6: Tomita's 3^{rd} DFA representation

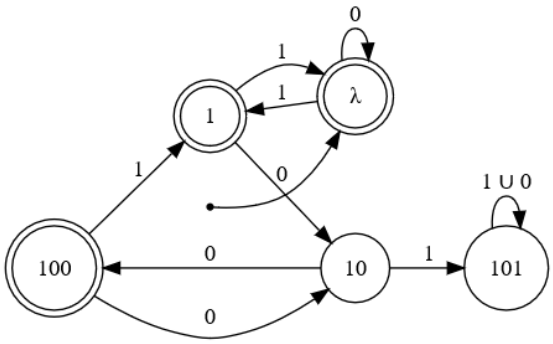

Figure 5.7: Tomita's 3^{rd} SFA representation

In the following figures"i" means *intervals*, and "e" means *equality*.

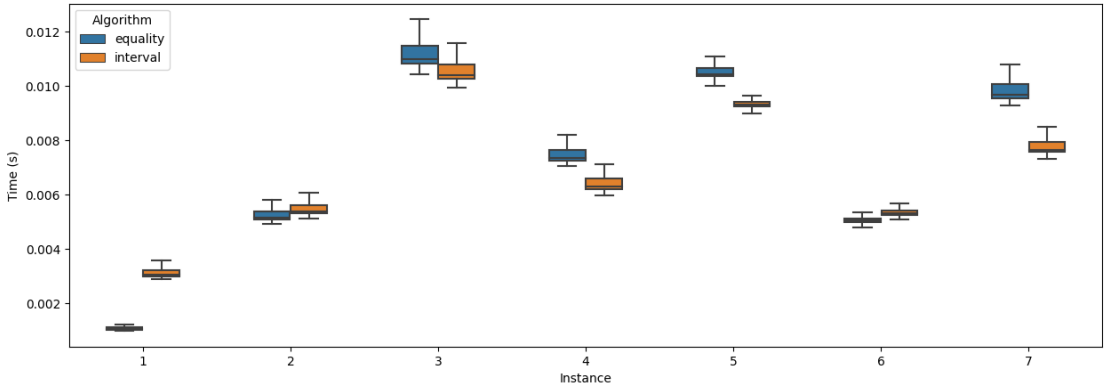

Figure 5.8: Time box plot

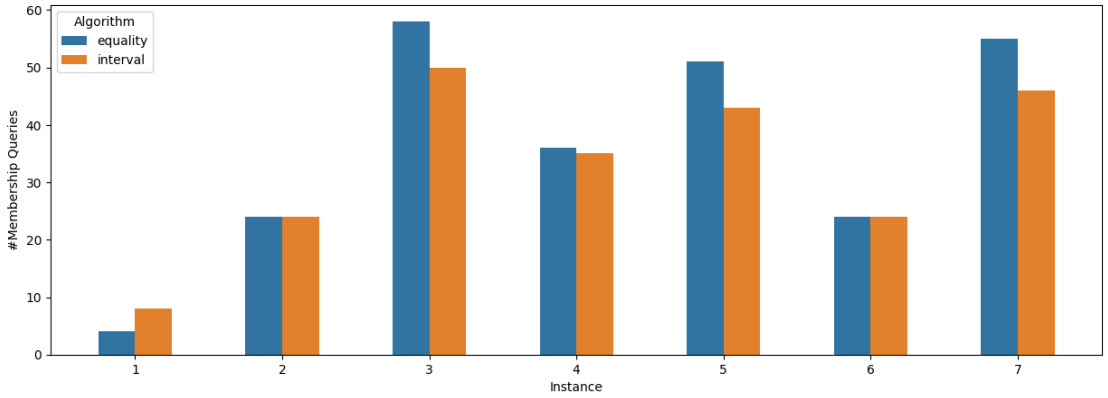

Figure 5.9: **MQ** count

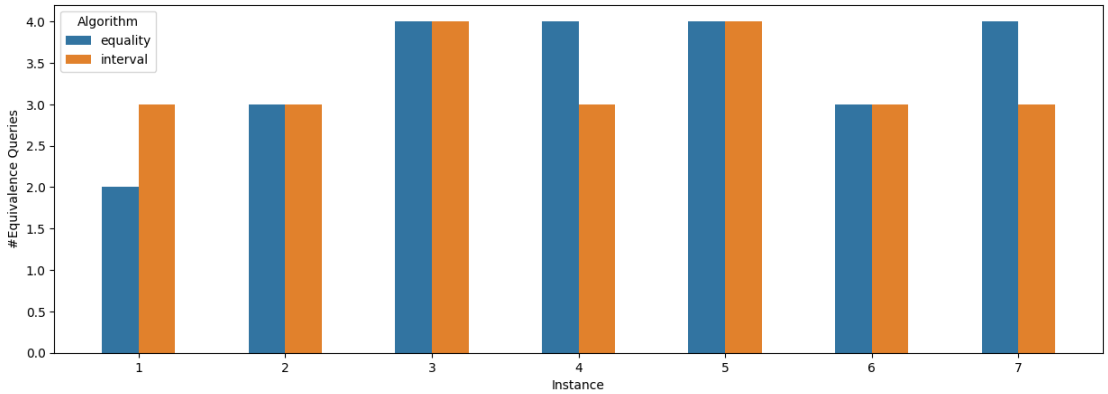

Figure 5.10: **EQ** count

There are no major differences between both Boolean algebra learners, but if we look carefully, there are two conclusions we might make. Even though we would need more data to verify it, it looks like given the same amount of **MQ** and **EQ**, equality algebra works faster than closed interval algebra. Yet, there are also instances where closed interval algebra is faster, we will expand on this point in the following example.

5.3 Example 3

Up until now, it is pretty obvious that Λ^* is quite slow in comparison to L^*. This makes a lot of sense, since as stated before, it does need to do more work in each iteration. But if one thinks about it, there should be some cases in which it should run faster. Lets say we run Λ^* with the automata we see in Figure 5.11. It could, in theory, just know the bounds of the intervals and no other symbol. So let us test this, we will generate random automata using the method described in [12] based on results from [13], and then *explode* the transitions, meaning transform one transition into n sequential transitions. As long as the **EQ** returns the correct counterexamples, Λ^* could potentially be significantly faster than L^*.

The automata to be tested are the ones in Figure 5.11 and Figure 5.13. Some other bigger automata were also tested, but since the results are maintained across the board, we will use these ones since they are smaller and thus easier to read.

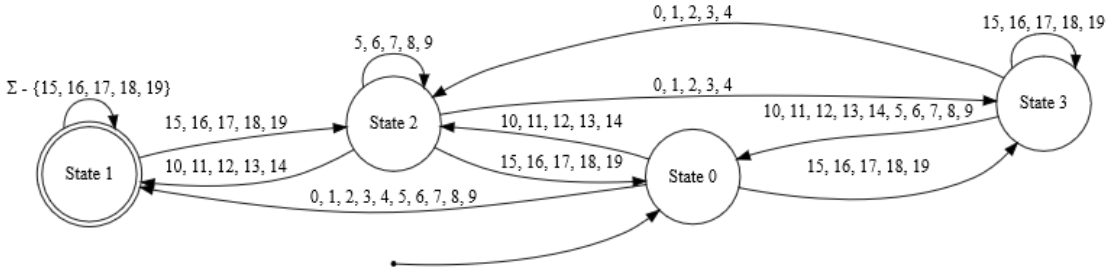

Figure 5.11: First exploded DFA

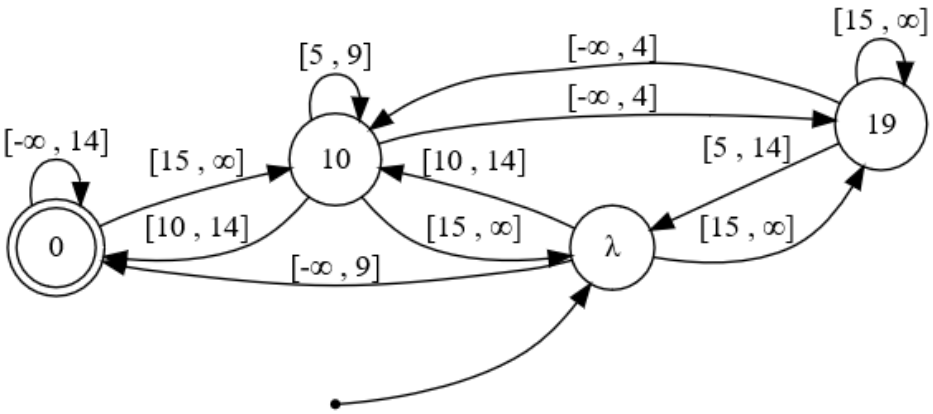

Figure 5.12: First exploded DFA representation as an SFA

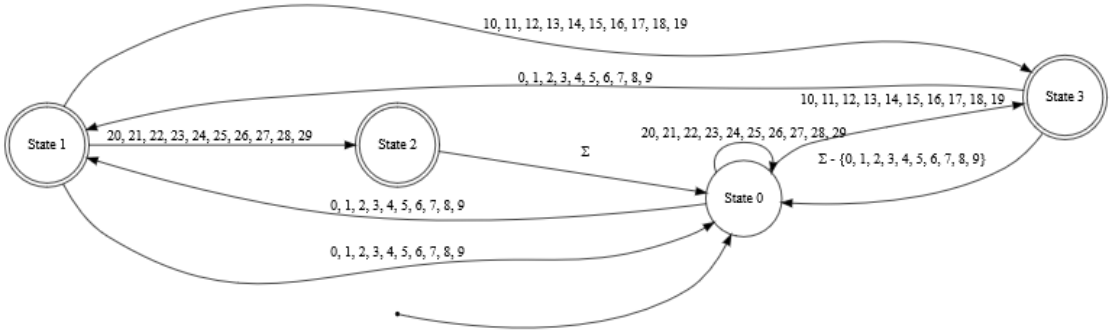

Figure 5.13: Second exploded DFA

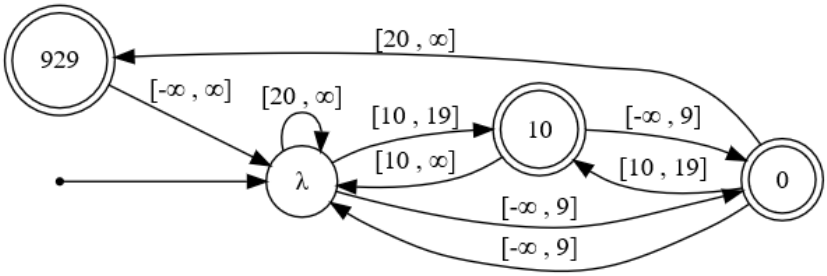

Figure 5.14: Second exploded DFA representation as an SFA

What we see in both cases in Figure 5.15 is that while the difference between both Boolean algebra learners is exacerbated, Λ^* cannot yet overcome the speed of L^*.

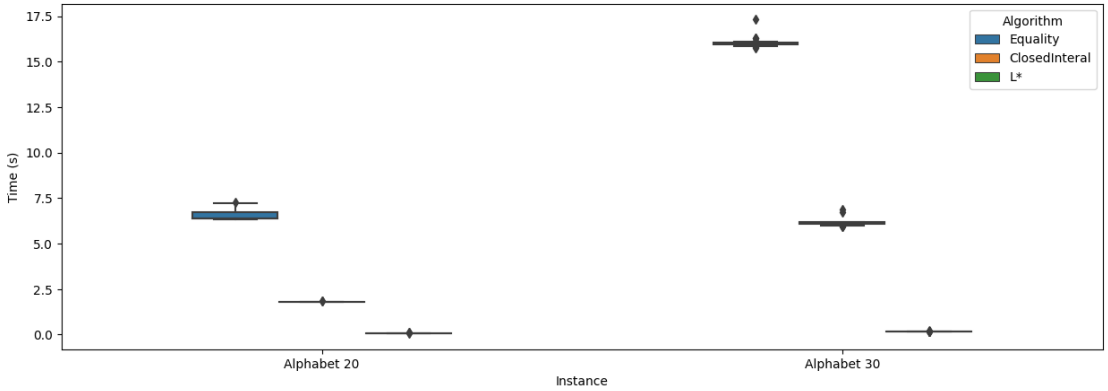

Figure 5.15: Time box plots

So we should look next at **MQ**, Figure 5.16, and **EQ**, Figure 5.17, quantity. What we see is that Λ^* is using significantly more **EQ**s than L^*. Why could this be happening? One possibility is that it might not be getting *good counterexamples*, so this would mean that it's working with not only an incomplete alphabet, but also it is getting useless information that does not allow it to learn the full intervals, but it learns it symbol by symbol.

There is a way we can check if this is the case, or at least to discard it in case it was not. We can look at the size of the *observed symbols* set. Luckily, the experiment already has this data, and indeed when we look at it, we can see that the size of this *observed symbols* set is the same as the DFA's alphabet ($|\sigma| = |\Sigma|$). So the question would be what would happen if the **EQ** was *optimal*, this is to be answered in future works.

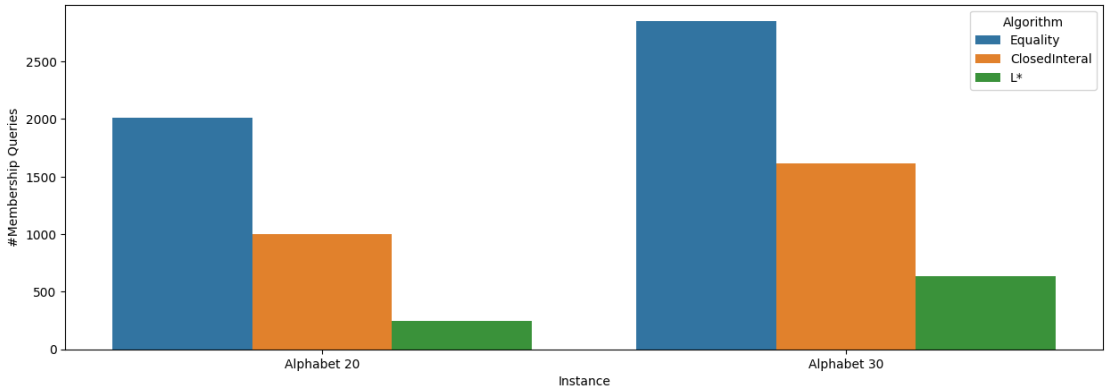

Figure 5.16: DFAs **MQ**

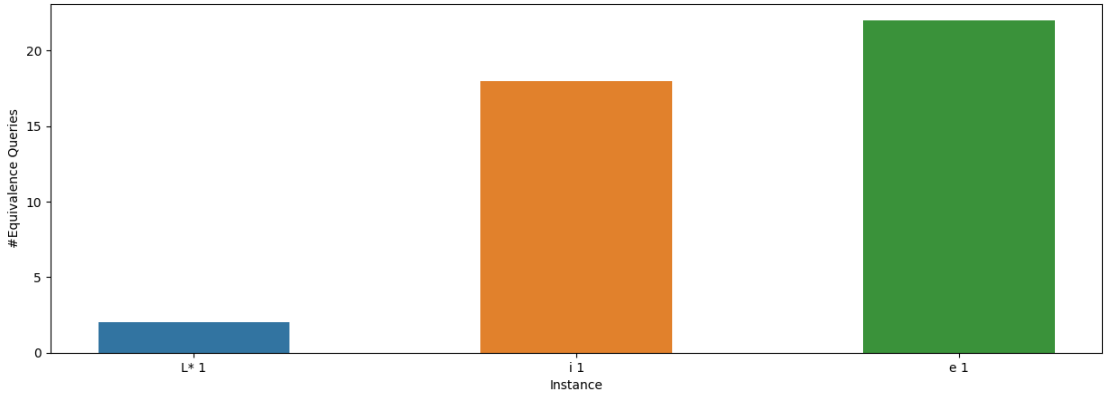

Figure 5.17: DFAs **EQ**

6 Conclusions

We presented an active learning algorithm for learning symbolic finite automata that are equivalent to a deterministic finite automaton. The algorithm was a slight variation of Drews and D'Antoni's Λ^*[4], in which the full alphabet is not necessary. This algorithm in itself being a variation of Angluin's L^*[3] algorithm and very similar to Mens and Maler's algorithm[14]. And we compared both this version of Λ^* and L^* algorithms, to see if and when they are to be used.

While we found that in all our case studies L^* was significantly better than Λ^*, this does not mean that Λ^* should not be used ever. We discussed how the bad results were in large part because of bad counterexamples, not because of the algorithm per se. So if we could have a comparison algorithm better suited to this case than the one described by J. E. Hopcroft, R. Motwani, and J. D. Ullman [5], we would get results faster than L^*, given big enough alphabets.

Another path is, to study how it works with infinite alphabets, since this variation of Λ^* does not need to know the full alphabet, it can be used for such regular grammars.

This last part could also be linked to the work by Mayr and Yovine[15], trying to aproximate a RNN representing an infinite alphabet under the PAC framework. This might solve part of the problem we faced with the comparison algorithm used in the **EQ**. Since the PAC framework chooses a counterexample arbitrarily instead of going in order like the one used in the experiments we ran.

All in all, despite not having the time efficiency results we expected, the work done is of great value and with different paths forward to work with this algorithm in the context of large alphabets. The work resulted in a whole new framework for development, in a state comparable to other published frameworks and easily extensible with new algorithms, as shown by the fact that it is now being used by different teams inside the university, like in the work done by Mayr, Yovine, Pan Basset and Dang [16].

7 References

[1] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[2] S. Shalev-Shwartz and S. Ben-David, "Understanding machine learning - from theory to algorithms." Cambridge University Press, 2014.

[3] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[4] S. Drews and L. D'Antoni, "Learning symbolic automata," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Legay and T. Margaria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 173–189.

[5] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

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

[7] K. Murphy, "Passively learning finite automata," Santa Fe Institute, Tech. Rep. 96-04-017, 1996.

[8] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, no. 3, pp. 302 – 320, 1978.

[9] L. D'Antoni and M. Veanes, "Minimization of symbolic automata," *SIGPLAN Not.*, vol. 49, no. 1, p. 541–553, jan 2014. [Online]. Available: https://doi.org/10.1145/2578855.2535849

[10] I.-E. Mens and O. Maler, "Learning regular languages over large ordered alphabets," 2015. [Online]. Available: https://arxiv.org/abs/1506.00482

[11] M. Tomita, "Dynamic construction of finite automata from examples using hill-climbing," in *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, Ann Arbor, Michigan, 1982, pp. 105–108.

[12] C. Nicaud, "Random deterministic automata," in *MFCS'14*. LNCS 8634, 2014, pp. 5–23.

[13] A. Carayol and C. Nicaud, "Distribution of the number of accessible states in a random deterministic automaton," *Leibniz Int. Proc. in Informatics*, vol. 14, pp. 194–205, 2012.

[14] O. Maler and I.-E. Mens, "Learning regular languages over large alphabets," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 485–499.

[15] F. Mayr and S. Yovine, "Regular inference on artificial neural networks," in *Machine Learning and Knowledge Extraction*, A. Holzinger *et al.*, Eds. Cham: Springer International Publishing, 2018, pp. 350–369.

[16] F. Mayr, S. Yovine, F. Pan, N. Basset, and T. Dang, "Towards efficient active learning of pdfa," 2022. [Online]. Available: https://arxiv.org/abs/2206.09004