Bachelor thesis
Computing Science


Radboud University

# A comparison of counterexample processing techniques in Angluin-style learning algorithms

*Author:*
Rick ten Tije
s1005826

*First supervisor/assessor:*
dr. J.C. Rot
jrot@cs.ru.nl

*Second assessor:*
prof. dr. F. W. (Frits) Vaandrager
F.Vaandrager@cs.ru.nl

March 31, 2022

## Abstract

Automata learning is the field of constructing finite-state models from observed input-output data. In this thesis the automata learning algorithms $L^*$ and $NL^*$ are discussed. $L^*$ learns a deterministic finite-state automaton (DFA) whereas $NL^*$ learns a residual finite-state automaton (RFSA). The algorithms will be compared by their performances on number of memberships queries, equivalence queries and automata sizes. Caching membership queries, averting consistency and Rivest-Schapire counterexample processing are improvements on these algorithms of which their effectiveness on $L^*$ and $NL^*$ in terms of membership queries, equivalence queries and automata sizes are evaluated.

# Contents

# Chapter 1

# Introduction

Automata theory is the field in computer science that studies abstract machines [8]. Many machines can be abstracted into finite state automata. Finite state automata can be used as a model for different kinds of hardware and software. A finite-state automaton consists of states and transitions. The states save important information about the history of the machine's execution. We only have a finite number of states, thus when constructing a finite state automaton, it is essential that the automaton only saves important information and discards unimportant information. What is important and what is not depends on the context.

It is sometimes possible to model a finite state automaton in a setting in which the inner workings of a system or machine is unknown, we call this setting a *black box* setting [5]. Under the black box setting, we can only interact with the machine by entering inputs and receiving outputs. We can often construct models from a machine under the black box setting. The field that focuses on learning such models is called *model learning* [19].

In this thesis, we will focus on a specific type of model learning, namely *automata learning* [1]. In automata learning, we investigate a machine under a black-box setting and try to model an automaton which represents the machine. There are many different types of automata. For automata learning, we will look at two types of automata: deterministic finite-state automata (DFAs) and nondeterministic finite-state automata (NFAs) [8]. We will also look at a subclass of NFAs namely residual finite-state automata (RFSAs) [6]. We will go deeper into the specifics of these different types of automata in the preliminaries chapter.

Automata learning is an active field. Active automata learning can be traced back as early as 1987 with Angluin's $L^*$ algorithm [1]. $L^*$ was the first algorithm that was able to actively learn automata, that is, $L^*$ was able to learn automata by actively experimenting on the to-be-learned automaton. Various studies followed improving $L^*$ [12, 18]; these improvements will be further discussed in Chapter 3. While $L^*$ focused on learning DFAs, a

succesful NFA automata learning algorithm was created in 1998 by Takashi Yokomori described in [21]. The advantage of NFAs compared to DFAs is that NFAs can contain less states compared to DFAs, this is explained more in depth in the preliminaries chapter of this thesis. A decade after Yokomori's NFA automata learning algorithm, an even more e cient NFA learning algorithm was created called *NL* [3]. *NL* is able to learn a subclass of NFAs, called RFSAs [6]. Two more recent DFA automata learning algorithms are the *TTT* algorithm [10] and the $L^{\#}$ algorithm [20]. A proper overview of model learning and automata learning can be found in [19].

The two automata learning algorithms we will take a closer look at in this thesis are *L* [1] and *NL* [3]. How these two di er will be explained further in the thesis. Both *L* and *NL* are able to learn a model by using *membership queries* and *equivalence queries*. The membership queries are used by the algorithm to nd the answer to the question: \Is this input accepted by the machine?" and the equivalence queries are used by the algorithm to nd the answer to the question: \Is this created model correct?". These queries are issued by the learning algorithm to the teacher. This teacher can be viewed as the representative of the system to be learned. The teacher knows the unknown language and thus can answer the membership and equivalence queries. We call this teacher the *minimally adequate teacher* [1]. We will compare these two algorithms in terms of membership queries, equivalence queries and model size.

*L* and *NL* can also be improved to reduce the number of membership queries and equivalence queries. In this thesis we will discuss the following optimizations: caching membership queries, averting consistency by treating counterexamples di erently [14] and Rivest-Schapire counterexample processing [18]. We will also show why the counterexample processing optimization by Rivest-Schapire will not work for NFAs.

Studies focusing on performance comparisons between automata learning algorithms and their variants in the literature are scarce. Certainly, most studies introducing new automata learning algorithms compare their results with other relevant algorithms; for example in [20] and [3]. The goal of this research is to measure the impact that the optimizations have on *L* and *NL* along with the relative performance of the algorithms themselves. Moreover, the comparison in this thesis can be used for future comparisons with other automata learning algorithms and optimization techniques.

In Chapter 2, I will explain some basic automata theory which is necessary to understand the successive chapters. In Chapter 3, I will introduce the algorithms *L* and *NL* in detail, and also the optimization techniques that can be used to improve these algorithms. In Chapter 4, the experiments and their results are described. In Chapter 5, I will describe some related work around automata learning. Chapter 6 concludes the research and describes future research directions.

# Chapter 2

# Preliminaries

In this chapter, I will explain some concepts which are important for understanding the research conducted in this thesis. Two basic concepts of automata theory are deterministic nite-state automata (DFAs) and nondeterminisitic nite-state automata (NFAs) [8]. We will also look at minimal automata and residual nite-state automata (RFSAs) [6].

## 2.1 Finite state automata

### 2.1.1 Deterministic finite-state automata

We rst de ne an alphabet $A$ as a nite set of letters. Sequences of these letters are called words. We de ne the set of all possible words made out of $A$ as $A$ . A language $L$ is a subset of $A$ ($L$ $A$ ). Note that the empty word, de ned by , is always in $A$ .

A deterministic nite-state automaton (DFA) is a nite-state machine that recognizes a certain language; it either accepts, or rejects a given word.

De nition 1. *A deterministic nite-state automaton is a 5-tuple* $M = (Q; A; ; q_0; F)$ *where* $Q$ *is the set of states,* $A$ *the alphabet,* $: Q \quad A \to Q$ *the transition function,* $q_0 \in Q$ *the initial state and* $F \quad Q$ *the set of nal states.*

Let us de ne a DFA $M_1$ where $Q = \{s_0; s_1; s_2\}$, $A = \{a; b\}$, $q_0 = s_0$, $F = \{s_2\}$ and its transition function as shown in Figure 2.2 in the form of a table. This DFA accepts the language of words that end with $ab$. We can represent a DFA as a graph. A graph representation of $M_1$ is given in Figure 2.1.

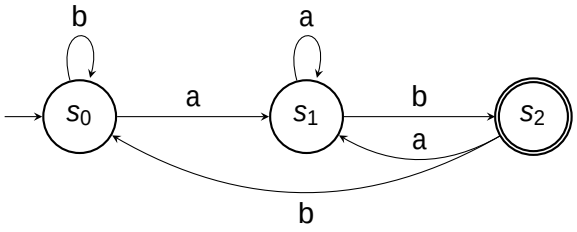


Figure 2.1: A graph representation of $M_1$

| | a | b |
|---|---|---|
| $s_0$ | $s_1$ | $s_0$ |
| $s_1$ | $s_1$ | $s_2$ |
| $s_2$ | $s_1$ | $s_0$ |

Figure 2.2: Transition table of $M_1$

The transition function takes a state and a letter as input and produces the resulting state. If we want to process a whole word instead of one letter we use the transition function for words denoted by .

De nition 2. *We de ne the transition function for words in a DFA M as Q A ! Q where Q is the set of states in M and A the set of words by means of induction:*

- *Base case: for all q 2 Q it holds that* $(q; ) = q$
- *Inductive case: for all q 2 Q; v 2 A and u 2 A it holds that* $(q; vu) = ( (q; v); u)$

To check if a word is accepted by a DFA, we use the transition function . If the resulting state of this function is in $F$, then the automaton accepts the given word. For example, the word *abb* is not accepted by $M_1$ since:

$$\begin{aligned} (s_0; abb) &= (( (s_0; a); bb) = (s_1; bb) \\ &= (( (s_1; b); b) = (s_2; b) \\ &= (( (s_2; b); ) = (s_0; ) \\ &= s_0 \end{aligned}$$

And because $s_0$ is not in $F$, the word *abb* is not accepted but the word *aab* is accepted since $(s_0; aab) = s_2$:

$$
\begin{aligned}
(s_0; aab) &= ((\ (s_0; a); ab) &&= (s_1; ab) \\
&= ((\ (s_1; a); b) &&= (s_1; b) \\
&= (\ (s_1; b);\ ) &&= (s_2;\ ) \\
&= s_2
\end{aligned}
$$

As stated before, DFAs accept a certain language. We de ne language acceptance for DFAs as follows.

De nition 3. *For a DFA $M = (Q; A; ; q_0; F)$, the language of state $q$ 2 $Q$ is de ned as: $L_q = fw\ j\ \ (q_0; w)$ 2 $Fg$. The language accepted by $M$ is the language of the initial state $L_{q_0}$.*

### 2.1.2 Nondeterministic finite-state automata

A nondeterminisitic nite-state automaton (NFA) has, contrary to a DFA, possibly multiple initial states and a slightly di erent transition function. Instead of transitioning to only one state with a given letter, the nondeterministic transition function can transition to multiple states with the same letter. This makes the transition function di erent from DFAs, instead of mapping to a single state as in our previous de nition: $Q\ \ A\ !\ Q$, we can now go from a state with an arbritary letter to possibly all other states so our transition function becomes of type: $Q\ \ A\ !\ 2^Q$ where $2^Q$ is the power set of $Q$.

We de ne NFAs as follows.

De nition 4. *A nondeterministic nite-state automaton is a 5-tuple $M = (Q; A; ; I; F)$ where $Q$ represents the set of states, $A$ the alphabet, the transition function $Q\ \ A\ !\ 2^Q$, $I\ \ Q$ the set of initial states and $F\ \ Q$ the set of nal states.*

We can also create an NFA for the language of words that end with *ab*. Let us call this NFA $M_2$ where $Q = fs_0; s_1; s_2g$, $A = fa; bg$, $I = fs_0g$ and $F = fs_2g$. A graphical representation of this NFA is shown in Figure 2.3 and the transition function for this NFA is shown in Figure 2.4 as a table.

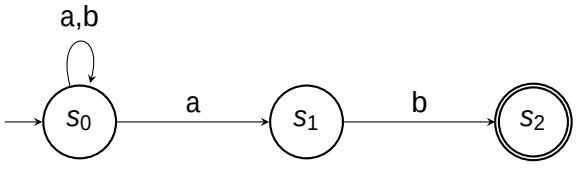


Figure 2.3: A graphical representation of $M_2$

| | a | b |
|---|---|---|
| $s_0$ | $\{s_0, s_1\}$ | $\{s_0\}$ |
| $s_1$ | $\emptyset$ | $\{s_2\}$ |
| $s_2$ | $\emptyset$ | $\emptyset$ |

Figure 2.4: Transition table of $M_2$

Similar to the transition function for DFAs, if we want to process a whole word instead of one letter we use the transition function for words denoted by . This function gets a state and a word as input and iterates through the word with the transition function and the current state producing a set of resulting states.

**Definition 5.** *We define the transition function for words in an NFA $M$ as $Q \times A^* \to 2^Q$ where $Q$ is the set of states in $M$ and $A$ the set of words by means of induction:*

- *Base case: for all $q \in Q$ it holds that $(q, \varepsilon) = \{q\}$*
- *Inductive case: for all $q \in Q$, $v \in A$ and $u \in A^*$ it holds that $(q, vu) = \bigcup_{q' \in \delta^*(q,v)} \delta(q', u)$*

Since produces a set of states for each iteration through a word, we will have to take the union of the next steps for the resulting set of states. The example below shows how the word *abb* is processed in $M_2$ with for NFAs.

$$\begin{aligned}
\delta^*(s_0, abb) &= \delta^*(\delta^*(s_0, a), bb) &&= \delta^*(\{s_0, s_1\}, bb) \\
&= \delta^*(\delta^*(s_0, b) \cup \delta^*(s_1, b)), b) &&= \delta^*(\{s_0\} \cup \{s_2\}, b) \\
&= \delta^*(\delta^*(s_0, b) \cup \delta^*(s_2, b)), \varepsilon) &&= \delta^*(\{s_0\} \cup \emptyset, \varepsilon) \\
&= \{s_0\}
\end{aligned}$$

Since $s_0$ is not in $F$, the word *abb* is not accepted by NFA $M_2$ but the word *aab* is accepted since $\delta^*(s_0, aab) = \{s_0, s_2\}$ and $s_2$ is a final state:

$$\begin{aligned}
\delta^*(s_0, abb) &= \delta^*(\delta^*(s_0, a), ab) &&= \delta^*(\{s_0, s_1\}, ab) \\
&= \delta^*(\delta^*(s_0, a) \cup \delta^*(s_1, a)), b) &&= \delta^*(\{s_0, s_1\} \cup \emptyset, b) \\
&= \delta^*(\delta^*(s_0, b) \cup \delta^*(s_1, b)), \varepsilon) &&= \delta^*(\{s_0\} \cup \{s_2\}, \varepsilon) \\
&= \{s_0, s_2\}
\end{aligned}$$

Finally, we can formally define language acceptance for NFAs.

**Definition 6.** *For an NFA $M = (Q, A, \delta, I, F)$, the language of states $Q' \subseteq Q$ is defined as: $L_{Q'} = \{w \mid \delta^*(q_0, w) \cap F \neq \emptyset\}$. The language accepted by $M$ is the language of the initial states $L_I$.*

### 2.1.3 Minimal automata

Two automata are equivalent if they accept the same language. A language can be represented by multiple di erent DFAs. A DFA is called minimal if there are no equivalent DFAs with fewer states. A regular language can always be represented by a unique minimal DFA [8]. The same cannot be said about NFAs [6].

For instance, take the language $L_1$ over the alphabet $\{a, b\}$ which contains all the words which have exactly one $b$ in them. Two graphical representations of DFAs are given in Figures 2.5 and 2.6. Both these DFAs accept $L_1$. However, the DFA in Figure 2.6 is the unique minimal DFA since there does not exist another DFA that accepts the same language with fewer states.

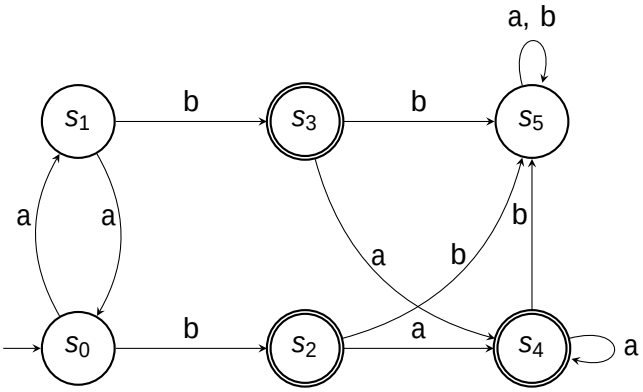


Figure 2.5: Nonminimal DFA of $L_1$

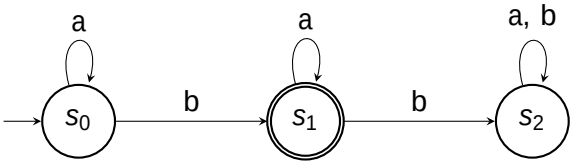


Figure 2.6: Minimal DFA of $L_1$

NFAs do not have the property of having a unique minimal automaton for every language. Take for example the language $L_2$ over the alphabet $\{a, b, c\}$ which contains two letter words in which each letter is di erent [2]. A minimal graphical representation of an NFA that accepts $L_2$ has ve states, but it is not unique as shown in Figure 2.7.

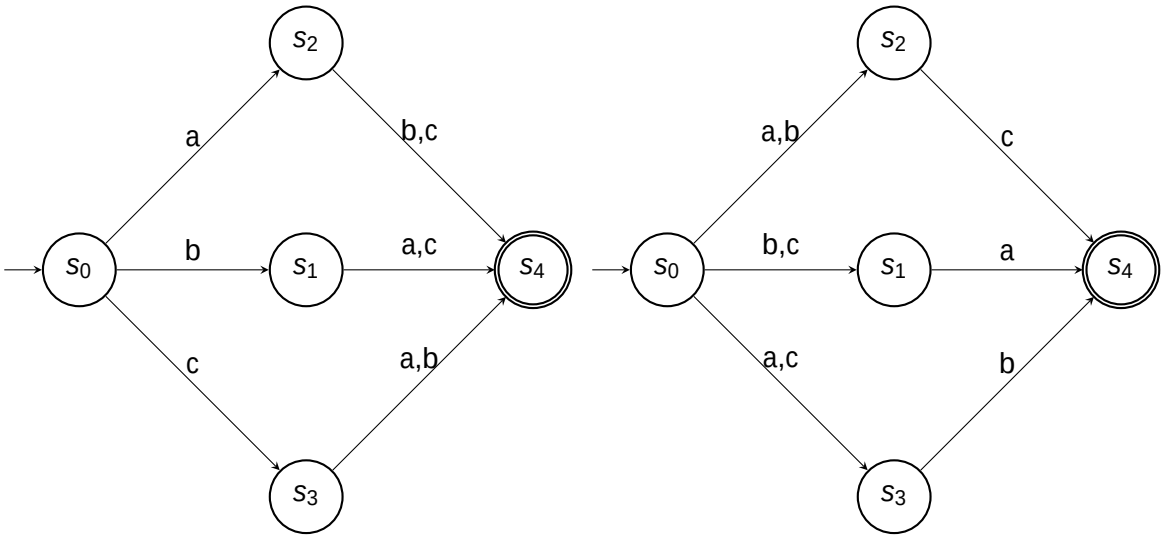


Figure 2.7: Two minimal representations of the NFA accepting $L_2$

### 2.1.4 Residual finite-state automata

As mentioned earlier, a property that NFAs do not have, contrary to DFAs, is the property that there is a unique minimal automaton for each regular language. Another property that DFAs have but NFAs do not always have is the property that every state of a DFA corresponds to a *residual language* [3].

De nition 7. *A language $L_r$ A is a residual language of another language L A if there is a word u 2 A such that $L_r$ = fv 2 A j uv 2 Lg. We denote the set fv 2 A j uv 2 Lg as u $^{1}$L. The set of residual languages of L is denoted by Res(L).*

There is a subclass of NFAs which does have the two properties of having a unique minimal automaton for each regular language and having each state of the automaton correspond to a residual language by de nition. This subclass of NFAs is called *residual nite state automata* (RFSAs) [6].

De nition 8. *A residual nite-state automaton is an NFA M: (Q;A; ;I;F) for which each q 2 Q, the language of that state $L_q$ is a residual language of the language of the automaton, or $L_q$ 2 Res(L(M)).*

Since RFSAs are nondeteministic, they can have less states than their corresponding minimal DFAs. Some states of a minimal DFA are super uous since they correspond to the union of languages of other states. To nd these super uous states we introduce *prime* and *composed* residual languages [3].

**Definition 9.** *A residual language $L_r$ of a language $L$ $A$ is called composed if there are other residual languages $L_1;:::;L_n$ 2 $Res(L)nL_r$ where $L_r$ = $L_1$ [ ::: [ $L_n$. If this does not hold, then $L_r$ is called prime.*

Each residual language represents a state in a RFSA. When we have a RFSA that consists only of states representing the prime residual languages we have a *canonical* RFSA. A canonical RFSA is also a minimal RFSA as shown in [6]. Since canonical RFSAs are minimal, they are a perfect candidate for learning as will be shown in the next chapter.

**Definition 10.** *For a regular language $L$ $A$ , the canonical residual finite-state automaton of $L$ is the NFA $M$: $(Q;A; ;I;F)$ where*

- *$A$ is the alphabet of $L$*
- *$Q$ is the set of prime residual languages of $L$.*
- *$I$ is the set of initial states consisting of prime residual languages included in $L$.*
- *$F$ is the set of final states which are the prime residual languages containing the empty word.*
- *is the transition function.*

Take again the language of words ending with $ab$ over the alphabet $fa;bg$ ($L = (a + b)\ ab$). The minimal DFA which accepts this language was shown in Figure 2.1 and a NFA representation of the same language was shown in Figure 2.3. The NFA representation is not a RFSA since the languages of states $s_1$ ($L_{s_1} = fbg$) and $s_2$ ($L_{s_2} = f\ g$) are not residual languages. The canonical RFSA of the same language is shown in Figure 2.8. The languages of the states in this RFSA are: $L_{s_0} = (a + b)\ ab$, $L_{s_1} = (a + b)\ ab +$ and $L_{s_2} = (a + b)\ ab + a\ b$. All these languages are residual languages since $L_{s_0} = \ ^1L$, $L_{s_1} = ab\ ^1L$ and $L_{s_2} = a\ ^1L$.

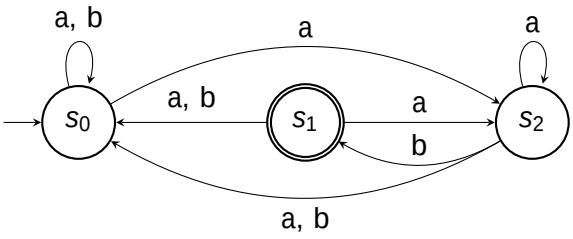


Figure 2.8: Canonical RFSA

# Chapter 3

# Automata Learning Algorithms

In this chapter, I will explain the algorithms *L** and *NL* in detail. For each algorithm, I will explain how it works followed by an example run through the algorithm. Moreover, optimization techniques such as caching membership queries, averting consistency and Rivest-Schapire counterexample processing will be explained and their function within *L** and *NL* will be made clear.

## 3.1 L*

In this section, *L** will be explained. *L** is an algorithm which can learn an unknown regular language by using queries and counterexamples [1]. The algorithm was created by Dana Angluin. Her algorithm serves as a foundation of other research related to learning these regular languages [20]. In the following subsections, I will explain the main idea of how Angluin's algorithm works.

### 3.1.1 Fundamentals

Suppose we have an unknown regular language *L* over a known alphabet *A*. Now the task at hand is to find an automaton that represents the same language as *L*. We do not know what *L* is but the *minimally adequate teacher* does. We can ask this teacher two things: if a word is in *L* (membership queries) and if a created conjecture equals *L* (equivalence queries). For the membership queries, the teacher responds with either a '+', indicating that a word is in *L*); or a '-', indicating the opposite. For the equivalence queries, the teacher either responds with *yes*, which means that the created automaton represents *L*; or with *no* and an accompanying counterexample, the counterexample distinguishes the created automaton and *L*. In other words,

the counterexample is either accepted by the created automaton but not in $L$, or is not accepted by the created automaton but is in $L$.

The algorithm uses *observation tables* to create an automaton. However, before we can de ne observation tables, we rst need to de ne *pre xes* and *su xes*. A pre x is a set of letters at the *beginning* of a word.

De nition 11. *A word p is a pre x of word w if there is a word u such that pu = w.*

When we talk about the pre xes of a word, we talk about all the subwords of that word which start at the beginning of the word. For example, the pre xes of the word aabb are aab, aa, a and .
A su x is a set of letters at the *end* of a word.

De nition 12. *A word s is a su x of word w if there is a word u such that us = w.*

For example, the su xes of the word aabb are abb, bb and b. Pre x-closed means that every pre x of all the members of the set is also a member of the set. Su x-closed means that every su x of all the members of the set is also a member of the set.

With pre xes and su xes de ned, we can now formally de ne observation tables.

De nition 13. *An observation table is a 3-tuple (S,E,row) where S is a nonempty nite pre x-closed set of strings, E is a nonempty nite su x-closed set of strings, and row a function de ned as*
*row*: $S \cup S A \to \{+, -\}^E$.

| $T_1$ | | a |
|---|---|---|
| | - | + |
| a | + | - |
| b | - | - |
| ba | - | - |
| aa | - | - |
| ab | + | - |
| bb | - | - |
| baa | - | - |
| bab | - | - |

Figure 3.1: An observation table.

We denote the upper part of an observation table by $O_{up}$ and the lower part by $O_{low}$.

An example of an observation table is given in Figure 3.1. In this observation table, the labels of the rows in $O_{up}$ represent $S = f\ ; a; b; bag$, the labels of the rows in $O_{low}$ represent all the words in the Cartesian product of $S$ and $A$ which are not already in $S$, and the column labels represent $E = f\ ; ag$. Each value in the table corresponds to a word which consists of a concatenation of the row label and the column label. *row* now maps the word to either a '+' or a '-' depending on the fact if the word is in the unknown language $L$. Whether a word is in $L$ is determined by asking a membership query for that word to the teacher. For example, according to $T_1$ the word aa is not in $L$ but the word ab is.

The algorithm can now create an automaton from this observation table if the table has the following two properties:

- Consistent, which means that if we take any two di erent words $s_1; s_2\ 2$ $S$ such that $row(s_1) = row(s_2)$, then for all $a$ in $A$ it must hold that $row(s_1\ \ a) = row(s_2\ \ a)$.
- Closed, which means that for each row in $O_{low}$, there should be an equal row in $O_{up}$. Thus for all $s_1\ 2\ S\ \ A$ there exists a $s_2\ 2\ S$ such that $row(s_1) = row(s_2)$.

We can see that $T_1$ is consistent since *row*(b) and *row*(ba) have the same value and both the values of *row*(ba) and *row*(baa) are the same and the values of *row*(bb) and *row*(bab) are the same.

We can also see that $T_1$ is closed since for each row in $O_{low} = ff\text{-},\text{-}g,\ f+,\text{-}g,\ f\text{-},\text{-}g,\ f\text{-},\text{-}g,f\text{-},\text{-}gg$ there is a row in $O_{up} = ff\text{-},+g,\ f+,\text{-}g,\ f\text{-},\text{-}g,\ f\text{-},\text{-}gg$ with the same value.

### Hypothesis construction

From a closed and consistent observation table, a DFA can be constructed. We call this constructed DFA a *hypothesis*.

**De nition 14.** *For an observation table O = (S,E,row) that is closed and consistent, a hypothesis* $M = (Q; A;\ ; q_0; F)$ *can be created where:*

- $Q = fr\ j\ r\ 2\ O_{up}g$*:*
- *A is the alphabet.*
- $(row(s); a) = q\ 2\ Q$ *where* $row(sa)\ 2\ q;$ *for* $s\ 2\ S$ *and* $a\ 2\ A$*:*
- $q_0 = q\ 2\ Q$ *where* $row(\ )\ 2\ q$*:*
- $F = fq\ 2\ Q\ j\ r(\ ) = +$ *for* $r\ 2\ qg$*.*

Closedness is necessary because of the way we de ned the transition function for the hypothesis. If a row in $O_{low}$ is not in $O_{up}$, then that row is not represented by a state and thus impossible to transition to. Consistency is necessary since multiple words in $S$ can represent the same state, and if we have a inconsistent table, then that would mean that we can transition to di erent rows and thus di erent states with the same word. This goes against the property of DFAs of having a unique transition from one state to another with the same word.

For example, in $T_1$ there are three distinct rows so we create three groups: $frow(\ )g$, $frow(\mathtt{a})g$ and $frow(\mathtt{b})$, $row(\mathtt{ba})g$; we rename these states to *q0*, *q1* and *q2* respectively for convenience. The initial state will be the group containing the empty word row, $row(\ )$. The nal states are the groups in which the column value of the empty word is a +. The transitions between the states can also be determined from the observation table. Suppose we are in state *q2*, if the next input is b, then we have a transition to state *q2* since the fact that we are in *q2* already tells us that reading a b should result in either $row(\mathtt{bb})$ or $row(\mathtt{bab})$ and both of have the same value of the group value of state *q2*. The resulting DFA can be seen in Figure 3.2. This DFA accepts the language of words which start with the letter *a* and end with any number of *b*'s.

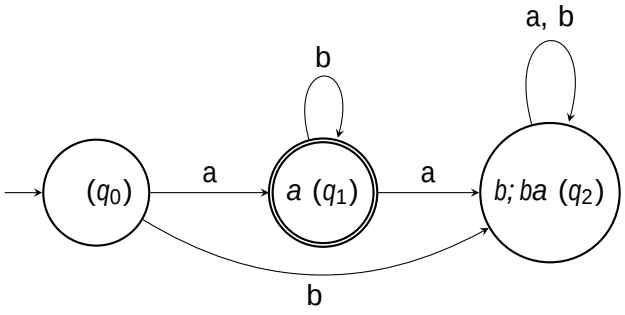


Figure 3.2: DFA of $T_1$

### Counterexamples

When the algorithm has created a hypothesis, it asks the teacher if the created automaton actually represents $L$. If the teacher replies with 'yes', the hypothesis correctly represents the unknown language and $L$ terminates. However, if the teacher replies with 'no', it also gives a counterexample. This counterexample is in $L$ and is not accepted by the hypothesis or the other way around. If a counterexample is given, the algorithm adds the counterexample and all its pre xes to $S$ and then it will start over again with making a new observation table and automaton. This process continues until an automaton is found that represents $L$.

### The algorithm

I will now give an overview of how $L^*$ exactly works by combining all the parts we discussed in this section.

$L^*$ starts by initializing $S$ and $E$ with the empty word. Then, it will create an initial observation table by using $S$, $E$, the alphabet $A$ and $row$. $L^*$ will then check for closedness and consistency. If the table is not consistent, then $L^*$ will acquire the word responsible for the inconsistency, add it to $E$ and update the observation table accordingly. If the table is not closed, then $L^*$ will add the word $sa \in S \cdot A$ for which $row(sa)$ is not in $O_{up}$ to $S$ and also update the observation table accordingly. After fixing the closedness and/or consistency violation, the algorithm repeats the process of checking for closedness and consistency violations. If the algorithm asserts that the observation table is closed and consistent, it will continue by creating a hypothesis. The hypothesis is given to the teacher in the form of an equivalency query. The teacher will then either respond with 'yes' or 'no'. If the teacher's respond is 'no', the given counterexample and its prefixes is added to $S$ and $L^*$ starts over again by checking for closedness and consistency violations. If the teacher's respond is 'yes', then the hypothesis represents the unknown language and the algorithm terminates.

A representation of how the algorithm works is also given in form of pseudo code below in Algorithm 1. In this pseudo code, the following pseudo functions are used:

- *membership*(), which asks membership queries for the given set of words and returns the corresponding values for each word.
- *createHypothesis*(), which creates a hypothesis with the given observation table.
- *equivalent*(), which issues an equivalence query to the teacher with the given automaton and returns a yes or no answer with an optional counterexample.
- *prefixes*(), which returns all the prefixes of a given word.

Algorithm 1 $L$

```
S ← {ε}
E ← {ε}
row ← membership((S ∪ S·A)·E)
O ← create initial observation table (S, E, row).
repeat
  while (S, E, row) not closed or not consistent do
    if (S, E, row) is not consistent then
      Find s1, s2 ∈ S, a ∈ A and e ∈ E such that row(s1) = row(s2)
      and T(s1·a·e) ≠ T(s2·a·e)
      E ← E + a·e
      row ← membership((S ∪ S·A)·E)
    end if
    if (S, E, row) is not closed then
      Find s1 ∈ S and a ∈ A such that row(s1·a) ≠ row(s) for all s ∈ S
      S ← s1·a
      row ← membership((S ∪ S·A)·E)
    end if
  end while
  M ← createHypothesis(S, E, row).
  Eq, counterExample ← equivalent(M)
  if Eq = False then
    S ← S + counterExample + prefixes(counterExample)
    row ← membership((S ∪ S·A)·E)
  end if
until Eq = True
Return M
```

### 3.1.2 A run through L

In this subsection we will do a run through $L$ to strengthen our understanding of the algorithm. In this run we will learn the DFA which accepts the language $L$ over the alphabet $A = \{a, b\}$ where $L = \{a, b\} \cup \{w \mid w \in A^* \text{ and } |w| > 2\}$. The algorithm starts by initializing $S$ and $E$ with the empty word $\varepsilon$. It then proceeds by making the initial observation table by asking the according membership queries and making sure the table is closed and consistent. The initial observation table is shown in Figure 3.3.

| $T_2$ | |
|---|---|
| | - |
| a | + |
| b | + |

Figure 3.3: Initial observation table $T_2$.

$T_2$ is is not closed, as $row(a)$ is in $O_{low}$ but not in $O_{up}$ so the algorithm will add $row(a)$ to $O_{up}$ and add all the necessary rows to $O_{low}$ using membership queries. The new observation table $T_3$ is shown in Figure 3.4.

| $T_3$ | |
|---|---|
| | - |
| a | + |
| b | + |
| aa | - |
| ab | - |

Figure 3.4: Observation table $T_3$.

$T_3$ is closed and consistent so $L$ now creates a hypothesis. We have two distinct rows in $O_{up}$ so the newly created automaton has two states. The transitions are created as described in the previous section. The resulting hypothesis is show in Figure 3.5.

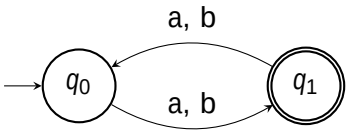


Figure 3.5: DFA of $T_3$

The algorithm continues to ask the teacher if the hypothesis actually represents $L$. The teacher replies with *no* and a counterexample. Let us assume that the teacher gave us the counter example aaaa. This is indeed a counterexample since it is in $L$ but the hypothesis does not accept it. The algorithm now adds the counterexample aaaa and its pre xes aaa and aa to $S$ and creates the new observation table $T_4$ depicted in Figure 3.6. Note that we do not add the pre xes a and to $S$ since they are already in $S$.

| $T_4$ | |
|---|---|
| | - |
| a | + |
| aa | - |
| aaa | + |
| aaaa | + |
| b | + |
| ab | - |
| aab | + |
| aaab | + |
| aaaab | + |
| aaaaa | + |

Figure 3.6: Observation table $T_4$.

$T_4$ is not consistent since $row(a) = row(aaa)$ but $row(aa) \neq row(aaaa)$. The algorithm fixes consistency violations by adding a new column with column label $a e$ and fill the new column by using membership queries. In this case, we got a inconsistency by adding the letter $a$ to the words $a$ and $aaa$, and the column which exposes the difference in the two rows is the column with column label . Thus, we should add $a$ to $E$ ($a = a$) and fill the newly created column by asking membership queries until we have the resulting table $T_5$ in Figure 3.7.

| $T_5$ | | a |
|---|---|---|
| | - | + |
| a | + | - |
| aa | - | + |
| aaa | + | + |
| aaaa | + | + |
| b | + | - |
| ab | - | + |
| aab | + | + |
| aaab | + | + |
| aaaab | + | + |
| aaaaa | + | + |

Figure 3.7: Observation table $T_5$.

The newly created observation table $T_5$ is still not consistent since $row(\ ) = row(aa)$ but if we add the letter $a$ to both words we get that $row(a) \neq row(aaa)$. The difference between $row(a)$ and $row(aaa)$ is visible in column with column label $a$. Thus, since adding the letter $a$ to the words and $aa$ results in different rows and since column label $a$ exposes the difference between the two rows we add $aa$ to $E$ and fill the newly created column by asking membership queries until we have the resulting table $T_6$ in Figure 3.8.

| $T_6$ | | a | aa |
|---|---|---|---|
| | - | + | - |
| a | + | - | + |
| aa | - | + | + |
| aaa | + | + | + |
| aaaa | + | + | + |
| b | + | - | + |
| ab | - | + | + |
| aab | + | + | + |
| aaab | + | + | + |
| aaaab | + | + | + |
| aaaaa | + | + | + |

Figure 3.8: Observation table $T_6$.

Table $T_6$ is closed and consistent so the algorithm makes a new hypothesis shown in Figure 3.9.

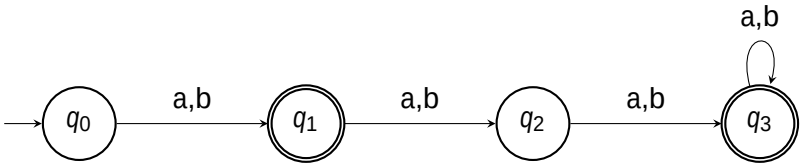


Figure 3.9: DFA of $T_6$

Finally, the algorithm asks the teacher if the hypothesis actually represents $L$. The teacher replies with *yes* and the program terminates.

### 3.1.3 Caching membership queries

A smart automaton learning algorithm needs to invoke the help of the teacher as little as possible. Caching the membership queries is an easy way of reducing the amount of membership queries needed to be asked to the teacher. Look for example at observation table $T_4$ in Figure 3.6. *row*(a ) = + since we issued a membership query to the teacher for the word a. Now consider table $T_5$ in Figure 3.7. In this table, a was added to $E$ after a consistency violation. $L$ would ask a membership query for each new cell created by the new column. For example, $L$ would ask a membership query for the word a to fill in the value for *row*( a) but we already know that *row*(a ) = and a = a so we do not need to ask a new membership query if $L$ remembered that we already queried that word.

Caching membership queries can reduce the total number of membership queries considerably. For this reason, and for the reason that it has no negative impact on other aspects of the algorithm, automata learning libraries such as `LearnLib` [16] cache membership queries by storing them in a hash table. I will also use this optimization in all experiments in Chapter 4.

### 3.1.4 Averting consistency

As we have seen in the previous section, inconsistency occurs when two rows in $O_{up}$, for example *row*($s_1$) and *row*($s_2$) where $s_1; s_2 \in S$ and $s_1 \neq s_2$, which have the same value, have a different value after you add the same letter from the alphabet to both $s_1$ and $s_2$. Inconsistency in an observation table can only occur when we add a counterexample and its prefixes to $O_{up}$. The only other operation that changes $O_{up}$ is the closedness operation and this only adds a row to $O_{up}$ which is not already in $O_{up}$, thus this cannot create a new situation where two rows in $O_{up}$ have the same value.

Thus, adding counterexamples to $S$ can create inconsistency. However,

we can avert this problem by changing the way we handle counterexamples [14]. Instead of adding a counterexample and all its prefixes to $S$, we add the counterexample and all its suffixes to $E$. This way, we do not increase the number of rows in $O_{up}$, we only increase the number of columns. As the observation table was already consistent before adding the counterexample, adding the counterexample and its suffixes to $E$ can only create more unique rows, not create more identical rows. This will ensure consistency within the table and thus a consistency check does not have to be executed. We call this variation of $L^*$ which adds counterexamples and their suffixes to $E$, $L^*_{col}$.

For example in our run through $L^*$, when we had the observation table in Figure 3.4 the teacher gave the counterexample **aaaa**. We added **aaaa** and its prefixes to $S$ and this already created an inconsistency in the table as shown in Figure 3.6. If we had added **aaaa** and its suffixes **aaa**, **aa** and **a** to $E$ we would have had the observation table in Figure 3.10.

| $T_4$ | | a | aa | aaa | aaaa |
|---|---|---|---|---|---|
| | - | + | - | + | + |
| a | + | - | + | + | + |
| b | + | - | + | + | + |
| aa | - | + | + | + | + |
| ab | - | + | + | + | + |

Figure 3.10: Observation table $T_4$.

After resolving two more closedness violations we end up with the table shown in Figure 3.11 which results in the same hypothesis as the hypothesis shown in Figure 3.9. Thus we can avert consistency checks altogether if we add counterexamples to $E$ instead of $S$.

| $T_5$ | | a | aa | aaa | aaaa |
|---|---|---|---|---|---|
| | - | + | - | + | + |
| a | + | - | + | + | + |
| aa | - | + | + | + | + |
| aaa | + | + | + | + | + |
| b | + | - | + | + | + |
| ab | - | + | + | + | + |
| aab | + | + | + | + | + |
| aaaa | + | + | + | + | + |
| aaab | + | + | + | + | + |

Figure 3.11: Observation table $T_5$ .

### 3.1.5 Smart counterexample processing

When an equivalence query is issued to the teacher, the teacher responds with a counterexample $z$ if the two given automata (the DFA $M$ representing the unknown language and the hypothesis $H$) are not equivalent. A counterexample shows that two automata are di erent since they give a different output for the same input. However, with just the counterexample, we do not know exactly what makes the two automata di erent. Since we only know that $z$ distinguishes $M$ and $H$, we add $z$ and all its su xes to the set $E$. After all, at some point in $H$ while processing the counterexample, $H$ has an incorrect transition which leads to the di erence to $M$. The problem with adding $z$ and its su xes to $E$ is that it makes the observation table a lot bigger and thus increases the number of membership queries considerably. It would be more e cient if we could nd the exact su x of the counterexample that is responsible for the di erence.

For example, take a look at the observation table in Figure 3.10. We can clearly see that the columns with column labels aaaa, aaa and a do not distinguish any rows in the table. For example, in $T_4$, $row(\ ) = row(\mathtt{aa})$ and $row(\ ) \not= row(\mathtt{a})$ and if we would delete all the columns with column labels aaaa, aaa and a, we would still have that $row(\ ) = row(\mathtt{aa})$ and $row(\ ) \not= row(\mathtt{a})$. However, the column with column label aa does change something. Su x aa distinguishes $row(\ )$ from $row(\mathtt{aa})$; these rows were the same in Figure 3.4. Thus adding the su xes aaaa, aaa and a to $E$ is obsolete.

More formally, when we look at the observation table of $H$, for $s_1; s_2 \in S$ and $a \in A$ where $row(s_1) = row(s_2\ a)$ the su x $e$ we need to expose the di erence should be such that $(q_0; s_1\ e) \not= (q_0; s_2\ a\ e)$. Here is the transition function for $H$ and $q_0$ is the initial state of $H$. But how do we

nd $e$?

The counterexample processing algorithm by Rivest-Schapire [18] does exactly what we described above. In their paper they describe their procedure of nding such a su x $e$ by using a kind of binary search. The process of nding the desired su x will now be explained in detail.

Suppose we have two DFAs $M_1$ and $M_2$ which are di erent, that is, they do not accept the same language. In context of $L$ , $M_1$ is the DFA representing the unknown language and $M_2$ is the hypothesis. Now assume that we have the counterexample $z$ which distinguishes these two machines. We start by checking if $z$ is accepted by $M_1$, denoted by $M_1{:}membership(z)$. If this is the case, we know it is not accepted by $M_2$ since $z$ is a counterexample and vice versa.

Now for every iteration we split $z$ in two parts. Let $u$ be the rst part of $z$ and $v$ the other part. We now put $u$ through $M_2$ and look in which state it ends. Recall that each state has its own pre x representing the state. We take the pre x of that state, we call this pre x $p$. Now take $p$, append $v$ to it and put it through $M_1$, denoted by $M_1{:}membership(pv)$. One of the following two things can now happen:

- $M_1{:}membership(pv) = M_1{:}membership(z)$, which means that $v$ or one of its su xes distinguishes $M_1$ and $M_2$. Since $v$ is shorter than $z$ (unless $u =$ ), you do not have to add the su xes of $u$ to $E$ which are not also a su x of $v$. Fewer words in $E$ means issuing fewer membership queries. However, we want to pinpoint the exact su x to add to $E$ so the next step of the algorithm is splitting $z$ at index $juj + \frac{1}{2}jvj$. This means that for the next iteration, $u$ will have more letters of $z$ and $v$ will have fewer letters of $z$ than the previous iteration.

- $M_1{:}membership(pv) \neq M_1{:}membership(z)$, which means that $v$ and all of its su xes do not distinguish $M_1$ from $M_2$. This means that a part of $u$ is needed to nd the correct su x. Thus, for the next iteration we split $z$ at index $\frac{1}{2}juj$.

The splitting works in a binary search-like way. So the rst split of $z$ will be at index $\frac{1}{2}jzj$. The second split, depending on $M_1{:}membership(pv)$ and $M_1{:}membership(z)$, will be at either $\frac{3}{4}jzj$ (in the rst case) or at $\frac{1}{4}jzj$ (in in the second case).

This whole process is repeated until we nd the exact su x $v$ of $z$ for which a one-letter longer su x or one-letter shorter su x of $z$ changes the outcome of $M_1{:}membership(pv) = M_1{:}membership(z)$ to $M_1{:}membership(pv) \neq M_1{:}membership(z)$ or vice versa. This su x $v$ is then our desired su x $e$ to add to $E$. Because of the binary search, we need $\log(jzj)$ membership queries to nd the one desired su x.

In $L$ the observation tables are lled by using membership queries. Thus, the number of membership queries needed to ll a table is the cardi-

nality of the table ((*S* [ *S* *A*) *E*). In *L* , the complexity of the number of membership queries is $O(kmn^2)$ where *k* is the length of the alphabet, *m* is the length of the longest counterexample received and *n* is the number of states of the minimal DFA representing the unknown language [1]. We can however improve the complexity of the number of membership queries by using the Rivest-Schapire counterexample processing technique described above. In the *L* variant which averts consistency checks described in section 3.1.4, *jEj* is bounded by $O(mn)$ but this can be improved to $O(n)$ although an additional *n log m* membership queries will be needed to achieve this [18]. This improves the complexity of the number of membership queries from $O(kmn^2)$ in *L* to $O(kn^2 + n \log m)$ in *L* which uses Rivest-Schapire counterexample processing.

The pseudocode of the algorithm can be found in Algorithm 2. In this pseudo code, the following pseudo functions are used:

- *membership*(), which asks membership queries for the given set of words and returns the corresponding values for each word.
- *len*(), which returns the length of the given word.
- *getEndState*(), which returns the state the hypothesis is in after transitioning through the hypothesis with a given word.

Algorithm 2 *Rivest Schapire counterexample processing*

```
cex counterExample
hypothesis createdAutomaton
unAut unknownAutomaton
cexOut unAut:membership(cex)
lower 1
upper len(cex) 2
while True do
  mid (lower + upper)==2
  endState hypothesis:getEndState(cex[: mid])
  prefix endState:pre x
  secHalfCex cex[mid :]
  mq unAut:membership(pre x + secHalfCex)
  if mq = cexOut then
    lower mid + 1
    if upper < lower then
      Return secHalfCex[1:]
    end if
  else
    upper mid + 1
    if upper < lower then
      Return secHalfCex
    end if
  end if
end while
```

## 3.2 NL*

When it comes to learning nite-state machines, DFAs are easier to understand and implement compared to NFAs. Learning a DFA with learning algorithms like *L* is possible since there is always a unique minimal DFA. This is not true for NFAs as we have shown in the preliminaries chapter. However, DFAs also have a huge disadvantage compared to NFAs. DFAs generally require more states than NFAs to represent the same regular language. With small regular languages this is not a problem but it becomes a signi cant problem if a DFA needs much more states to represent a language. *L* can not handle NFAs but in recent years, researchers have constructed algorithms that are able to learn NFAs. In this section I will describe such an algorithm, namely *NL* .

The algorithm *NL* is an automaton learning algorithm which is able to learn certain non-deterministic nite state automata in an *L* -like manner [3]. *NL* is able to learn residual nite state automata (De nition 8) which are non-deterministic nite state machines with some DFA properties. Most notably, every regular language has a unique minimal RFSA.

*NL* is not the rst algorithm which learned NFAs as this was also done in [21]. However, in [21], the resulting NFAs were non-canonical RFSAs and thus could have more states than their corresponding minimal DFAs. This issue is addressed in *NL* which always results in a canonical RFSA.

### 3.2.1 Learning RFSAs

As stated before, *NL* learns a RFSA in an *L* -like manner. *NL* also makes use of observation tables but there are some crucial di erences. Closedness and consistency of an observation table is determined di erently yet the general idea is the same. An observation table in *NL* has so called *prime rows*. Before we explain what a *prime row* is, we have to de ne the join operation for observation table rows.

De nition 15. *A join of two rows of an observation table, $r_1 \sqcup r_2$, results in a single row in which each value is determined by taking the join of the pair of values from both $r_1$ and $r_2$ where $\sqcup$ = and $+ \sqcup + = +$ $\sqcup + = +$ $+ \sqcup + = +$.*

For example, in Figure 3.12, *row*(aa) is the join of *row*( ) and *row*(a).

De nition 16. *A row r in an observation table O (r 2 rows(O)) is a prime row if and only if $r \not\subseteq r_1 \sqcup r_2$ where $r_1; r_2$ 2 rows(O) and $r_1 \not\subseteq r_2$. A prime row is marked with *.*

Prime rows play a big role in ensuring RFSA-closedness whereas in RFSA-consistency, prime rows do not play a part. However, to explain both RFSA-closedness and RFSA-consistency, we rst have to introduce a new operation, the subset relation for rows .

| | $T_7$ | | a | b |
|---|---|---|---|---|
| * | | + | - | - |
| * | a | - | + | - |
| * | b | - | - | - |
| | aa | + | + | - |
| * | ab | - | - | - |
| * | ba | - | + | - |
| * | bb | - | - | - |

Figure 3.12: An observation table in $NL$.

**Definition 17.** *A row $r_1 \in rows(O)$ is called a subset of another row $r_2 \in rows(O)$ if for each value of $r_1$, $v_1 \in r_1$, and the corresponding value of $r_2$, $v_2 \in r_2$, it holds that $v_1 = + \Rightarrow v_2 = +$. A row is called a strict subset if $r_1 \neq r_2$ and the aforementioned statement holds. The subset relation is denoted by $\subseteq$ and the strict subset relation is denoted by $\subset$.*

For example, in Figure 3.12, $row(ba) \subset row(aa)$ and $row(b)$ is a subset of all other rows in the table.

Recall that in $L^*$ an observation table was considered closed if each row in $O_{low}$, which represented the rows in the lower part of the table, was also present in $O_{up}$, the upper part of the table. RFSA-closedness is not as strict as closedness in $L^*$ algorithm because, compared to DFAs, you can have more transitions from a single state with the same letter in RFSAs; this will be explained in more detail further on in this section.

**Definition 18.** *An observation table $O$ is RFSA-closed if for each row $r \in O_{low}$, $r = \bigcup \{r' \in O_{up\_primes} \mid r' \subset r\}$.*

RFSA-closedness is not the same as closedness in $L^*$. As we can see in Figure 3.12, $T_7$ is RFSA-closed since all the rows in $O_{low}$ are composed of prime rows of $O_{up}$. But the table is not closed in the sense of $L^*$, as $row(aa)$ is not present in $O_{up}$.

As for consistency, recall that in $L^*$ an observation table was consistent if we could take any two rows in $O_{up}$, $row(s_1)$ and $row(s_2)$, which have the same value, then for all $a \in A$ it must hold that $row(s_1 a) = row(s_2 a)$. RFSA-consistency works a little differently.

**Definition 19.** *An observation table $O$ is RFSA-consistent if any two rows in $O_{up}$, $row(s_1)$ and $row(s_2)$, where $row(s_1) \subseteq row(s_2)$, it holds that for all $a \in A$ $row(s_1 a) \subseteq row(s_2 a)$.*

RFSA-consistency is not the same as consistency in $L$ . Take for example the table in Figure 3.13, this table is consistent in context of $L$ since $row(\ ) = row(a)$ and both $row(a) = row(aa)$ and $row(b) = row(ab)$ but the table is not RFSA-consistent as $row(a)$ $row(b)$ but $row(ab)$ $6$ $row(bb)$.

| | $T_7$ | | a |
|---|---|---|---|
| * | | - | - |
| * | a | - | - |
| * | b | + | + |
| * | aa | - | - |
| * | ab | + | + |
| * | ba | + | - |
| * | bb | - | - |

Figure 3.13: RFSA-inconsistent observation table

### Hypothesis construction

Besides the di erence in closedness and consistency checks between $NL$ and $L$ , the creation of a hypothesis from an observation table does also work somehow di erently compared to $L$ .

De nition 20. *From an observation table O(S,E,row) which is RFSA-closed and RFSA-consistent, an NFA M = (Q,I,F, ) can be created by de ning:*

- $Q = O_{up\_primes}$
- $I = fr\ 2\ Q\ j\ r \quad row(\ )g$
- $F = fr\ 2\ Q\ j\ r(\ ) = +g$
- $(row(s);a) = fr\ 2\ Q\ j\ r \quad row(sa)g$ *for* $s\ 2\ S$ *with* $row(s)\ 2\ Q$ *and* $a\ 2\ A$

All the rows in the upper part of the observation table which are prime will represent a state in the automaton. The initial states are the states for which any of their respective rows are a subset of the empty word row. The nal states are the states for which any of their respective rows have a + as their empty word column value.

Since with NFAs, you can have multiple destinations from one state with the same letter, the transition function will not result in a single state, but a set of states. Thus, the transition function for $NL$ will take a state

(which represents rows in $O_{up\_primes}$) and a letter, it then appends the letter $l$ to the row label $r$ that state represents and look in $O$ what value the resulting rows represent $row(rl)$. The destination of the transition function will now be all the states for which $row(rl)$ is a subset of.

For example, in Figure 3.12, if we want to transition from the state which represent $row(a)$ with the letter $b$, the resulting states will be the states of which $row(ab) = f\ ;\ ;\ g$ will be a subset of; in this case that will be all other states since $f\ ;\ ;\ g$ is a subset of all rows in $O_{up\_primes}$.

Remark 1. *It is possible to omit RFSA-consistency from the algorithm since this will still result in a canonical RFSA. However, removing RFSA-consistency does have a negative impact on the e ciency of the algorithm. Take for example the observation table in Figure 3.16. This table is RFSA-closed but not RFSA-consistent since row( ) row(a) but row(a) 6 row(aa). If we ignore RFSA-consistency and wanted to create a hypothesis from this table, we get the hypothesis shown below in Figure 3.14.*

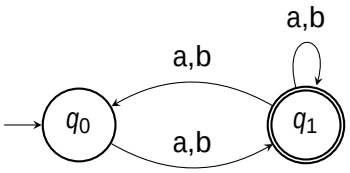


Figure 3.14: A hypothesis

*The hypothesis shown in Figure 3.14 is not consistent with regard to the table in Figure 3.16 since the word $aa$ is accepted by the hypothesis but is not in the language according to the table. This problem is however easily solved by adding the counterexample and its su xes to $E$. However, this does require an extra equivalence query and this can be avoided by checking for RFSA-consistency.*

### Counterexamples

When *NL* has a RFSA-closed and RFSA-consisent table and a hypothesis has been created, the algorithm will ask the teacher if the created automaton correctly represents the unknown language. Just like in $L$ , the teacher will either respond with yes or no. If the teacher replies with yes, we are done and the algorithm terminates. And if the teacher responds with no, it will also give an additional counterexample which distinguishes the language which the created automaton represents from the unknown language. When given a counterexample, the algorithm will add the counterexample and all its su xes to $E$ and runs the algorithm over again with making a new observation table and automaton. This process continues until an automaton is found that represents the unknown language.

### The algorithm

I will now give an overview of how *NL* exactly works by combining all the parts we discussed in this section.

*NL* starts by initializing $S$ and $E$ with the empty word. Then, it will create an initial observation table by using $S$, $E$, the alphabet $A$ and *row*. *NL* will then check for RFSA-closedness and RFSA-consistency. If the table is not RFSA-closed, then *NL* will add the word $sa \in S \cdot A$ for which *row*($sa$) cannot be composed of prime rows in $O_{up}$ to $S$ and also update the observation table accordingly. If the table is not RFSA-consistent, then *NL* will acquire the word responsible for the inconsistency, add it to $E$ and update the observation table accordingly. After fixing the RFSA-closedness and/or RFSA-consistency violation, the algorithm repeats the process of checking for RFSA-closedness and RFSA-consistency violations. If the algorithm asserts that the observation table is RFSA-closed and RFSA-consistent, it will continue by creating a hypothesis. The hypothesis is given to the teacher in the form of an equivalency query. The teacher will then either respond with 'yes' or 'no'. If the teacher's respond is 'no', the given counterexample and its suffixes are added to $E$ and *NL* starts over again by checking for RFSA-closedness and RFSA-consistency violations. If the teacher's respond is 'yes', then the hypothesis represents the unknown language and the algorithm terminates.

Note that *NL* uses the approach of adding counterexamples to $E$ instead of $S$ as discussed in a previous section about averting consistency. Moreover, *NL* cannot use the treatment of adding counterexamples to $S$ as done in $L^*$, as this leads to a non-terminating algorithm [3].

A representation of how *NL* works is also given in form of pseudo code below in Algorithm 3. In this pseudo code, the following pseudo functions are used:

- *membership*(), which asks membership queries for the given set of words and returns the corresponding values for each word.
- *createHypothesis*(), which creates a hypothesis with the given observation table.
- *equivalent*(), which issues an equivalence query to the teacher with the given automaton and returns a yes or no answer with an optional counterexample.
- *prefixes*(), which returns all the prefixes of a given word.

Algorithm 3 *NL*

```
S    f g
E    f g
row    membership((S [ S  A)  E)
O    create initial observation table (S, E, row).
repeat
  while (S, E, row) not RFSA-closed or not RFSA-consistent do
    if (S, E, row) is not RFSA-closed then
      Find s1 2 S and a 2 A such that row(s1 a) 2 O_prime_low n O_prime_up
      S    s1 a
      row    membership((S [ S  A)  E)
    end if
    if (S, E, row) is not RFSA-consistent then
      Find s1;s2 2 S; a 2 A and e 2 E such that row(s1)    row(s2) and
      T(s1 a e) 6 T(s2 a e)
      E    E + a e
      row    membership((S [ S  A)  E)
    end if
  end while
  M    createHypothesis(S;E;row).
  Eq; counterExample    equivalent(M)
  if Eq = False then
    E    E + counterExample + =mathitprefixes(counterExample)
    row    membership((S [ S  A)  E)
  end if
until Eq = True
Return M
```

### 3.2.2 A run through NL

For this run we will learn a RFSA with *NL* which accepts the language $L = \{a;bg [ \{w \mid w 2 A$ and $|w| > 2g$ over the alphabet $A = \{a;bg$ which we also used in the $L$ run in section 3.1.2. Like in $L$ , the algorithm starts by initializing $S$ and $E$ with the empty word . It creates an observation table with $S$, $E$, *row* and the alphabet and ensures that it is RFSA-closed and RFSA-consistent. If the table is not RFSA-closed, it will add all the prime rows which make the table not closed from the lower part of the table to the upper part of the table and update the table accordingly. If the table is not RFSA-consistent, it will add a new column with as label the string $a$ $e$ (where $a 2 A$ and $e 2 E$) responsible for the inconsistency. The initial observation table for this run is shown in Figure 3.15

| | $T_8$ | |
|---|---|---|
| * | | - |
| * | a | + |
| * | b | - |

Figure 3.15: Initial observation table $T_8$.

$T_8$ is not RFSA-closed, as $row(a)$ can not be composed of rows in $O_{up\_primes}$ so the algorithm will add $row(a)$ to $O_{up}$ and add all the necessary rows to $O_{low}$ using membership queries. The new observation table $T_9$ is shown in Figure 3.16.

| | $T_9$ | |
|---|---|---|
| * | | - |
| * | a | + |
| * | b | + |
| * | aa | - |
| * | ab | - |

Figure 3.16: Observation table $T_9$.

$T_9$ is not RFSA-consistent, as $row(\ ) \neq row(a)$ but $row(a) \neq row(aa)$ so the algorithm will add $a$ to $E$ and fill the newly created column using membership queries. The new observation table $T_{10}$ is shown in Figure 3.17.

| | $T_{10}$ | | a |
|---|---|---|---|
| * | | - | + |
| * | a | + | - |
| * | b | + | - |
| * | aa | - | + |
| * | ab | - | + |

Figure 3.17: Observation table $T_{10}$.

$T_{10}$ is RFSA-closed and RFSA-consistent, so the algorithm will create a hypothesis. The hypothesis is shown in Figure 3.18.

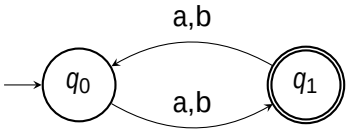


Figure 3.18: Hypothesis for $T_{10}$

Then the algorithm asks the teacher if the hypothesis actually represents $L$. The teacher replies with *no* and a counterexample. Let us assume that the teacher gave us the counter example aaaa. This is indeed a counterexample since it is in $L$ but the hypothesis does not accept it. The algorithm now adds the counterexample aaaa and its su xes aaa and aa to $E$ and extends the observation table to $T_{11}$ depicted in Figure 3.19.

| | $T_{11}$ | | a | aa | aaa | aaaa |
|---|---|---|---|---|---|---|
| * | | - | + | - | + | + |
| * | a | + | - | + | + | + |
| * | b | + | - | + | + | + |
| * | aa | - | + | + | + | + |
| * | ab | - | + | + | + | + |

Figure 3.19: Observation table $T_{11}$.

$T_{11}$ is not RFSA-closed, as $row(aa)$ can not be composed of rows in $O_{up\_primes}$ so the algorithm will add $row(ab)$ to $O_{up}$ and add all the necessary rows to $O_{low}$ using membership queries. The new observation table $T_{12}$ is shown in Figure 3.20.

| | $T_{12}$ | | a | aa | aaa | aaaa |
|---|---|---|---|---|---|---|
| * | | - | + | - | + | + |
| * | a | + | - | + | + | + |
| * | aa | - | + | + | + | + |
| * | b | + | - | + | + | + |
| * | ab | - | + | + | + | + |
| | aaa | + | + | + | + | + |
| | aab | + | + | + | + | + |

Figure 3.20: Observation table $T_{12}$.

$T_{12}$ is RFSA-closed and RFSA-consistent, so the algorithm will again

create a hypothesis. The hypothesis is shown in Figure 3.21.

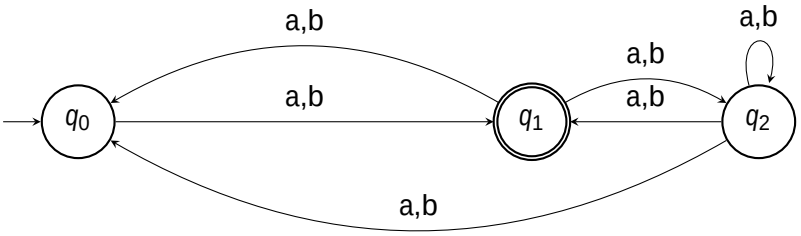


Figure 3.21: Hypothesis of $T_{12}$

Finally, the algorithm asks the teacher if the hypothesis actually represents $L$. The teacher replies with *yes* and the program terminates with the hypothesis as the resulting RFSA.

In both the run through $L$ and the run through $NL$, we learned the language $L = \{a,b\} \cup \{w \mid w \in A \text{ and } |w| > 2\}$. Notice that the canonical RFSA learned by $NL$ in Figure 3.21 is one state smaller than the corresponding minimal DFA learned by $L$ in Figure 3.9.

### 3.2.3 Nondeterministic Rivest-Schapire

We have discussed the smart counterexample processing technique by Rivest-Schapire [18] in context of deterministic learning. However, in $NL$ we are dealing with nondeterministic automata. In $L$ we only dealt with DFAs, as the automaton representing the unknown language and the hypothesis were both deterministic finite-state automata. With $NL$, the created hypothesis is a RFSA.

Recall the idea behind the technique described in section 3.1.5. When we look at a observation table $H$, for $s_1, s_2 \in S$ and $a \in A$ where $row(s_1) = row(s_2 a)$ the suffix $e$ we need to expose the difference should be such that $(q_0, s_1 e) \neq (q_0, s_2 a e)$. Here $\delta$ is the transition function for $H$ and $q_0$ is the initial state of $H$. Finding $e$ is the goal of the algorithm.

To find $e$ we needed to find the exact transition in the hypothesis which led to the difference between the hypothesis and the DFA representing the unknown language. However, with $NL$ the hypothesis is a RFSA and thus every state can have multiple transitions with the same letter. This makes finding the difference between the hypothesis and the DFA representing the unknown language difficult. In the next paragraph, I will explain where Rivest-Schapire's counterexample processing algorithm would fail in the context of nondeterministic automata learning.

Recall that the processing algorithm split the counterexample $z$ into two parts. The second part of $z$, $u$, was put through the hypothesis to obtain the prefix of the state in which $u$ ended. If the hypothesis is a DFA, we

will always end up in the same state when putting the same word through the DFA and thus we would obtain a single pre x representing the end state. But if the hypothesis is a RFSA, we can end up in multiple states representing multiple di erent pre xes. This makes determining the exact di erence between the hypothesis RFSA and the DFA representing the unknown language near impossible. For this reason, I could not implement Rivest-Schapire counterexample processing in *NL* for more experiments.

# Chapter 4

# Experiments

In this chapter, we compare the different versions of $L^*$ and $NL^*$. To achieve this, we need code of the algorithms and a set of regular languages to learn.

I implemented $L^*$ and $NL^*$ myself according to their description in [1] and [3]. Writing the code myself instead of using other automata learning libraries such as `LearnLib` [16] or `libalf` [4] gave me a thorough understanding of the algorithms and it made making changes to these algorithms, such as implementing additional optimizations, easier. Writing the code for the algorithms myself also made the results of the experiments consistent as I used the same implementation decisions, such as data structures, when writing the algorithms. A random walk equivalence oracle from the library AALpy [15] was used as the equivalence oracle. Besides the equivalence oracle, their DFA generation function was also used to obtain the necessary minimal DFAs to represent the regular languages for the experiments. All the code used for these experiments is written in Python.

I generated 2600 random DFAs ranging from 1 to 200 states over a 2-letter alphabet with AALpy's DFA generation function. These random DFAs were first tested for minimality before being used in the experiments. This amount of DFAs comes close to the amount of languages used in [3]; there 3180 regular languages were used for which their minimal DFAs also ranged from 1 to 200 states.

I will compare the following versions of the algorithms:

- $L^*$ which adds counterexamples and their prefixes to $S$ ($L^*$).
- $L^*$ which adds counterexamples and their suffixes to $E$ ($L^*_{col}$).
- $L^*$ which uses Rives-Schapire counterexample processing to add only a single suffix to $E$ for each counterexample ($L^*_{RS}$).
- $NL^*$ which adds counterexamples and their suffixes to $E$ ($NL^*$).
- $NL^*$ without RFSA-consistency ($NL^*_{no\_cons}$).

For all these versions, I compared the number of membership queries and number of equivalence queries issued to the teacher. A comparison of the number of states in the resulting automata was also made (only relevant for *NL* versions). As mentioned earlier in section 3.1.3, all versions contain the caching membership optimization.

Unfortunately, we do not have a lot of data regarding the *NL* experiments. This is mainly due to time constraints and implementation decisions. In my implementation of the algorithm, the resulting RFSA first needs to be determinized before it can be checked for equivalence by the teacher. Determinizing an NFA takes a substantial amount of time which increases exponentially by the size of the NFA. For this reason and due to time constraints, I could not obtain the results for regular languages which can be represented by a minimal DFA with 67 or more states. Thus in the following sections, the *NL* experiments will have a smaller sample size compared to the *L* experiments.

In section 4.1, I will compare the number of membership queries between the algorithms and their variants. After that, we will look at the comparison between the algorithms in terms of equivalence queries in section 4.2. Finally, in section 4.3 I will compare the algorithms in terms of resulting automaton size.

## 4.1 Membership queries

In Figure 4.1, the results of the membership queries experiments on *L* and its variants are shown. These graphs show the number of membership queries needed (y-axis) before learning minimal DFAs of various number of states (x-axis). For example, you will need around 2300 membership queries to learn a minimal DFA of 100 states with *L* . Note that the classic *L* algorithm performs better and produces more consistent results in terms of membership queries compared to $L_{col}$. However, when we apply Rivest-Schapire counterexample processing to $L_{col}$, we get slightly better results compared to classic *L* and significant better results compared to $L_{col}$ without Rivest-Schapire counterexample processing. The reason $L_{col}$ performs poorly is because the equivalence oracle, while not often, can return long counterexamples and the membership query complexity of $L_{col}$ linearly depends on the length of the longest counterexample [18]. The Rivest-Schapire counterexample processing technique does not have this dependency and thus performs significantly better.

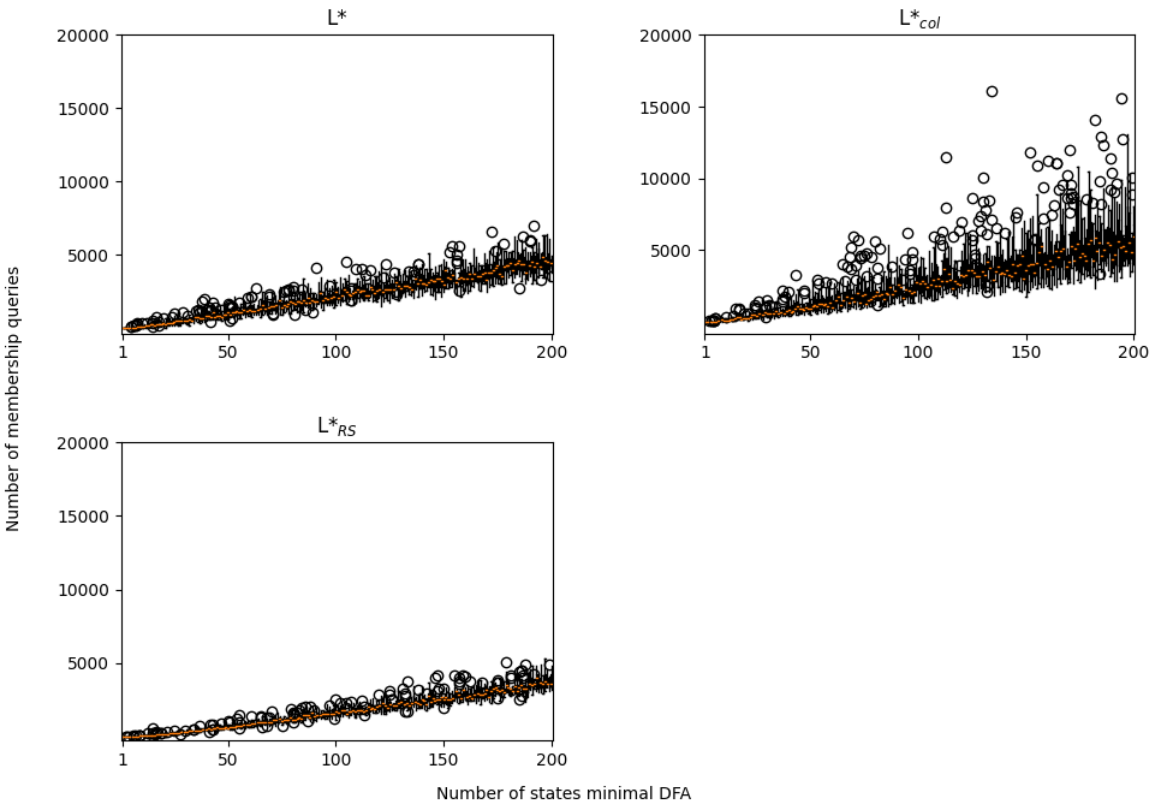


Figure 4.1: $L$ membership queries comparison

In Figure 4.2, the results of the membership queries experiments on $NL$ and $NL$ without RFSA-consistency are shown. These graphs show the number of membership queries needed (y-axis) before learning canonical RFSAs from minimal DFAs of various number of states (x-axis). For example, when given a minimal DFA of 25 states, $NL$ will need around 600 membership queries to learn a canonical RFSA accepting the same language. As we can observe from the graphs, RFSA-consistency does have little to no impact on the number of membership queries required to learn a canonical RFSA of a regular language.

Compared to the graphs for $L$ in terms of membership queries, $NL$ performs worse which is remarkable since it contradicts the results in [3]. However, since our sample size is smaller for $NL$ , it might be that $NL$ performs better in bigger resulting automata as also shown in [3]. For these experiments however, $NL$ performs similar to the $L_{col}$ variant without Rivest-Schapire counterexample processing in terms of membership queries.

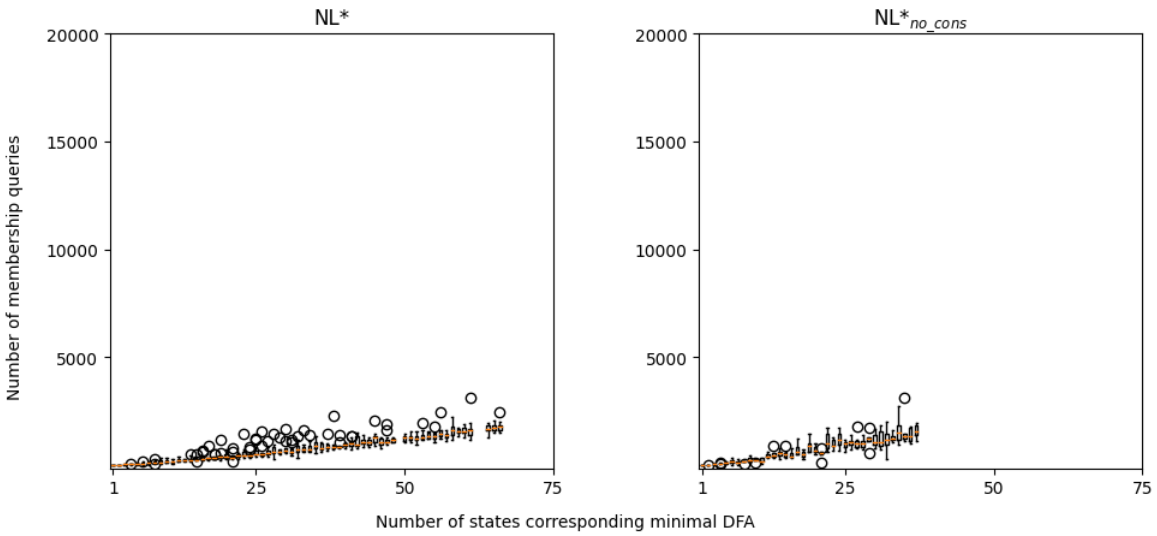


Figure 4.2: *NL* membership queries comparison

## 4.2 Equivalence queries

In Figure 4.3, the results of the equivalence queries experiments on $L^*$ and its variants are shown. These graphs show the number of equivalence queries needed (y-axis) before learning minimal DFAs of various number of states (x-axis). From these results, we can conclude that the classic $L^*$ algorithm performs similar to the $L^*_{RS}$ variant in terms of equivalence queries. Remarkably, Rivest-Schapire counterexample processing has a negative impact on the $L^*_{col}$ variant in terms of equivalence queries needed.

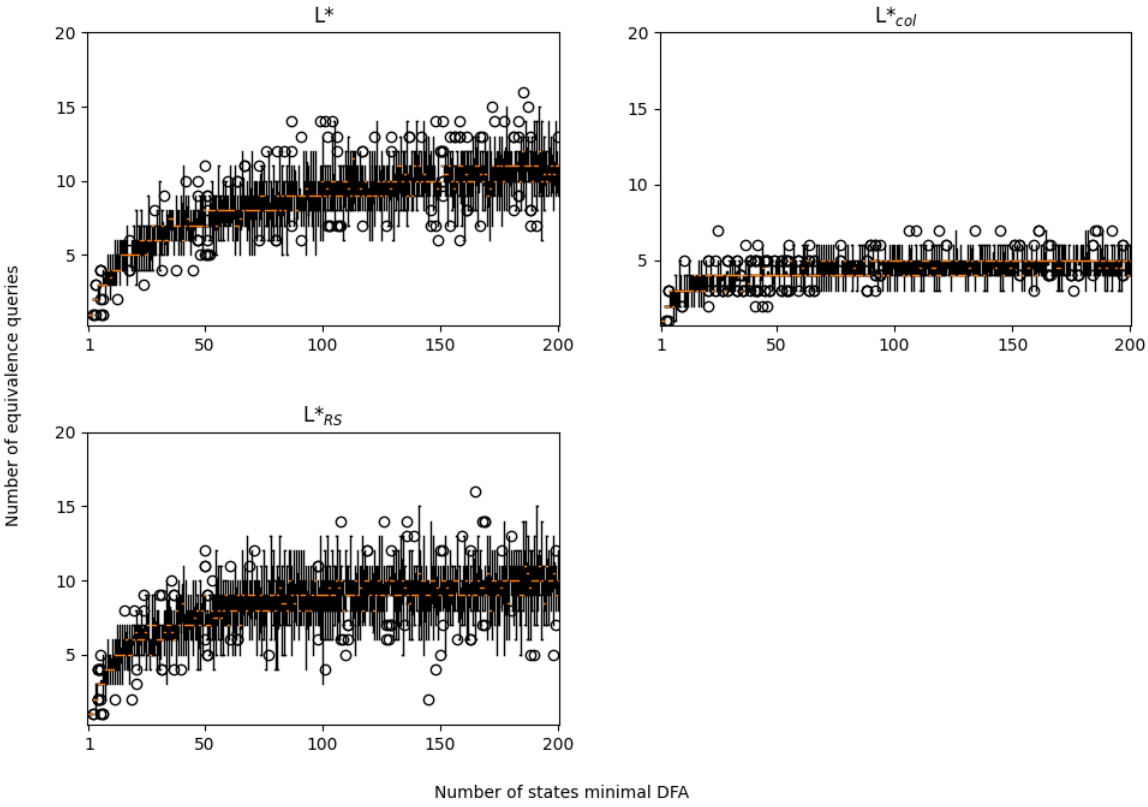


Figure 4.3: $L^*$ equivalence queries comparison

In Figure 4.4, the results of the equivalence queries experiments on $NL^*$ and its variants are shown. These graphs show the number of equivalence queries needed (y-axis) before learning canonical RFSAs from minimal DFAs of various number of states (x-axis). The $NL^*$ algorithm performs best out of all the other variants of learning algorithms experimented on in this thesis in terms of equivalence queries. On average, three equivalence queries are needed to learn a canonical RFSA with their corresponding minimal DFA ranging from 1 to 66 states.

Contrarily, $NL^*$ without RFSA-consistency seems to perform the worst out of all the other variants of learning algorithms experimented on in this thesis in terms of equivalence queries.

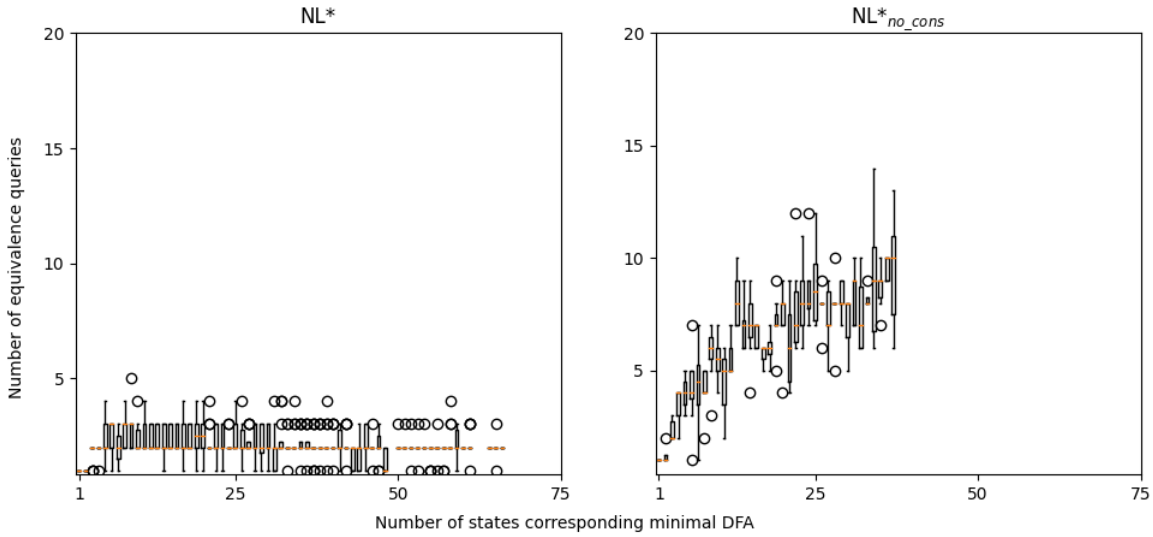


Figure 4.4: *NL* equivalence queries comparison

## 4.3 Automata sizes

To acknowledge the full potential of *NL* , consider the following DFAs which accept the following language [13]:

$L_i$ = *fw j w 2 fa;bg* and the i-th letter from the right is an *ag*

To create a minimal DFA of $L_i$ you would need $2^i$ states whereas the minimal amount of states needed to create a RFSA of $L_i$ is $i + 1$. This means that for $i = 5$, there are already 32 states needed for a minimal DFA to accept $L_5$ and only six states are needed for a canonical RFSA to accept $L_5$. This exponential increase in size for DFAs and linear increase in size for RFSAs is shown in Figure 4.5.

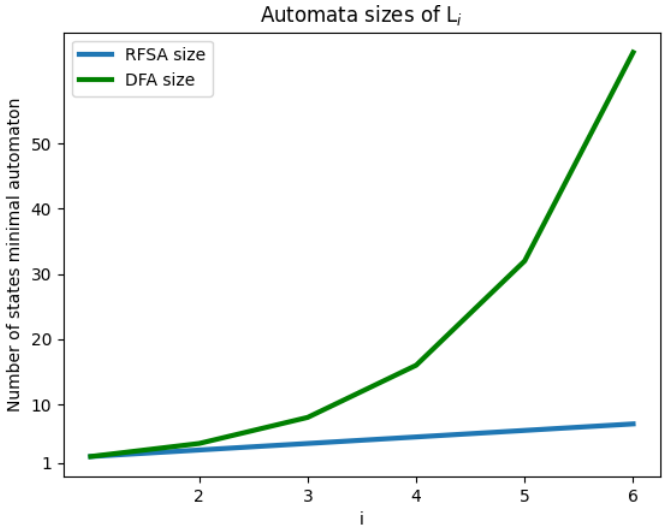


Figure 4.5: $L_i$ automata size comparison

Thus *NL* would perform extraordinary well in terms of automata size on languages for which their canonical RFSAs can be explicitly smaller than their corresponding minimal DFAs. In the next paragraph, we will look at how *NL* performs on my randomly generated minimal DFAs.

In Figure 4.6, the results of the automata size experiments on both *L* and *NL* are shown. This graph shows the number of states of the automaton learned by the algorithms (y-axis) compared to the number of states of the corresponding minimal DFA (x-axis). Since *L* and its variants learn the minimal DFA which they receive as input, their learned automaton has the same number of states as their input DFA. In *NL* , the learned automaton is a canonical RFSA and thus can have less states than the minimal DFA the algorithm received as input. Both *NL* and *NL* without RFSA-consistency perform slightly better than *L* and its variants in terms of learned automaton size. RFSA-consistency does not appear to have any in uence on the resulting RFSA size as it should not by de nition of the learned canonical RFSA.

The results obtained here are di erent compared to the results obtained in [3]. *NL* seems to perform slightly better in [3] in terms of resulting automaton size. However, due to the small sample size, the full potential of *NL* with more complex regular languages cannot be shown as was claimed in [3]. One more explanation as to why my resulting canonical RFSAs have more states compared to the learned canonical RFSAs in [3], is that I used totally randomly generated minimal DFAs as inputs for the algorithm whereas in [3], they used a di erent process of generating the large set of regular languages that they experimented on which is described in [7].

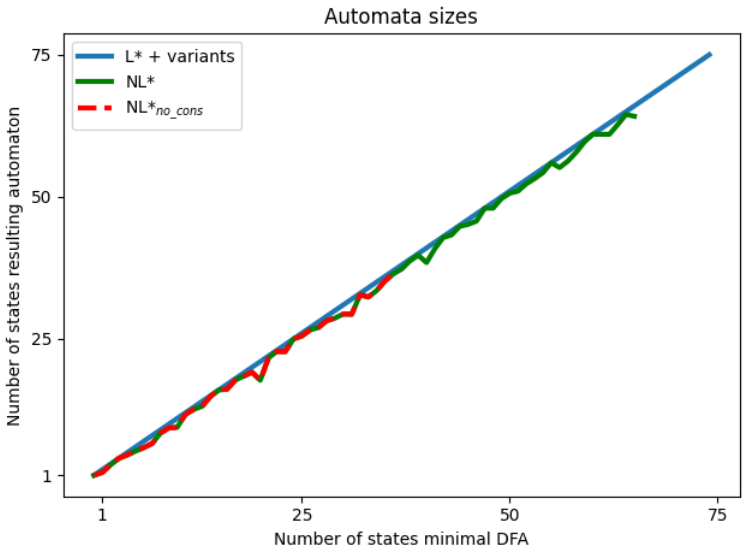


Figure 4.6: Automata size comparison

# Chapter 5

# Related Work

Over the last decades, numerous automata learning algorithms were created. Active automata learning started with Angluin's $L^*$ [1] algorithm which is able to learn determinsitic finite-state automata. The years thereafter, researchers have found ways to improve $L^*$ with optimizations like Rivest-Schapire counterexample processing [18] and changing the treatment of counterexamples [14]. Shortly after, nondeterministic automata learning algorithms were constructed [21]. $NL^*$ followed which was essentially an improved version of Yokomori's [21] nondeterministic automaton learning algorithm. More recent automata learning algorithms are the $TTT$ algorithm [10] from 2014 and the $L^\#$ algorithm [20] from 2021. In short, automata learning algorithms are of broad and current interest. As stated before, a proper overview of model learning and automata learning can be found in [19].

Although there is abundant literature on these different automata learning algorithms, literature focused on how these algorithms compare with each other, and more specifically their variants, is scarce. Nonetheless, a counterexample analysis for active automata learning algorithms can be found in [11] and an evaluation of practical performance on active automata learning algorithms can be found in [9]. However, these comparisons do not include nondeterministic learning algorithms such as $NL^*$. The research in this thesis gives the reader a more in-depth understanding of two relevant automata learning algorithms, $L^*$ and $NL^*$, along with their differences. Moreover, the research gives an elaborate explanation on some interesting optimization techniques used in these algorithms, most notably, Rivest-Schapire counterexample processing. It also gives the reader an idea on how these algorithms and their optimized variants compare with each other by showing a comparison between the two algorithms in terms of membership queries, equivalence queries and automata sizes.

# Chapter 6

# Conclusions

In conclusion, we explored the theory behind the learning algorithms $L$ and $NL$ which gave us an understanding about their usage and capabilities. We also looked at optimization techniques such as caching membership queries, averting consistency and Rivest-Schapire counterexample processing which can be used on these algorithms. We learned that Rivest-Schapire counterexample processing is effective in minimizing the amount of membership queries but it worsened the amount of equivalence queries. We also established the importance of RFSA-consistency as it had little to no impact on the number of membership queries, but a significant impact on the number of equivalence queries. Regarding resulting automata size, $NL$ does not seem to perform much better compared to $L$ when used on randomly generated minimal DFAs. However, $NL$ can result in much smaller automata compared to $L$ when used on suitable languages as we have seen in section 4.3.

## 6.1 Future work

In this thesis, the automata learning algorithms $L$ and $NL$ were discussed along with three optimizations. For future research, other automata learning algorithms like the TTT algorithm [10] or the $L^{\#}$ algorithm [20] can be compared with the results obtained from this research. For these future comparisons, experiments on the number of membership queries and equivalence queries can be performed similar to the experiments in this thesis. Moreover, other possible optimizations such as enforcing consistency and closure strategies as described in [17] can be implemented in these automata learning algorithms which in turn can also be compared with the results from this research. From these comparisons, possible ideas for new and more efficient automata learning algorithms can be formed.

# Bibliography


[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87{106, 1987.

[2] Andre Arnold, Anne Dicky, and Maurice Nivat. A note about minimal non-deterministic automata. *Bull. EATCS*, 47:166{169, 1992.

[3] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of NFA. In *IJCAI*, pages 1004{1009, 2009.

[4] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 360{364. Springer, 2010.

[5] Mario Bunge. A general black box theory. *Philosophy of Science*, 30(4):346{358, 1963.

[6] Francois Denis, Aurelien Lemay, and Alain Terlutte. Residual nite state automata. In *STACS*, volume 2010 of *Lecture Notes in Computer Science*, pages 144{157. Springer, 2001.

[7] Francois Denis, Aurelien Lemay, and Alain Terlutte. Learning regular languages using rfsas. *Theor. Comput. Sci.*, 313(2):267{294, 2004.

[8] John E. Hopcroft and Je rey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[9] Malte Isberner. *Foundations of active automata learning: an algorithmic perspective*. PhD thesis, Technical University Dortmund, Germany, 2015.

[10] Malte Isberner, Falk Howar, and Bernhard Ste en. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV*, volume 8734 of *Lecture Notes in Computer Science*, pages 307{ 322. Springer, 2014.

[11] Malte Isberner and Bernhard Ste en. An abstract framework for counterexample analysis in active automata learning. In *ICGI*, volume 34 of *JMLR Workshop and Conference Proceedings*, pages 79{93. JMLR.org, 2014.

[12] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[13] Mark V. Lawson. Finite automata. In *Handbook of Networked and Embedded Control Systems*, pages 117{144. Birkhauser, 2005.

[14] Oded Maler and Amir Pnueli. On the learnability of in nitary regular sets. *Inf. Comput.*, 118(2):316{326, 1995.

[15] Edi Muskardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. Aalpy: An active automata learning library. In *ATVA*, volume 12971 of *Lecture Notes in Computer Science*, pages 67{ 73. Springer, 2021.

[16] Harald Ra elt and Bernhard Ste en. Learnlib: A library for automata learning and experimentation. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 377{380. Springer, 2006.

[17] Harald Ra elt, Bernhard Ste en, Therese Berg, and Tiziana Margaria. Learnlib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11(5):393{407, 2009.

[18] Ronald L. Rivest and Robert E. Schapire. Inference of nite automata using homing sequences. *Inf. Comput.*, 103(2):299{347, 1993.

[19] Frits W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86{95, 2017.

[20] Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wi mann. A new approach for active automata learning based on apartness. *CoRR*, abs/2107.05419, 2021.

[21] Takashi Yokomori. Learning non-deterministic nite automata from queries and counterexamples. In *Machine Intelligence 13*, pages 169{ 189. Clarendon Press, Oxford, 1992.