MASTER INFORMATIQUE

UNIVERSITÉ CÔTE D'AZUR

Master in Computer Science

Course : TER

Comparison of L* and NL* algorithm

Author:
Fissore Davide

Supervisor:
Cinzia Di Giusto
Etienne Lozes

Abstract

Angluin's L* algorithm [1] is a famous algorithm for learning deterministic finite state automata from membership and equivalence queries. Bollig et al NL* [4] algorithm is a variant of this algorithm, where the automaton that is learned may be non deterministic, and therefore exponentially smaller. The goal of this TERD is to understand the theory behind these two algorithms, to implement and compare them.

Spring 2021-2022

Contents

1 Introduction

Humans have the ability to learn. This is the ability to understand theoretical information or practical behaviors from different situations. This phase of learning is usually possible by performing some actions and then registering the consequence.

An example may be a *vending machine* in a foreign country. If you want to get a cup of coffee from that machine and you don't know ow it works, the way to get your cup is to press the keys randomly and every time you get an answer from the machine, you can understand if the sequence of actions you performed is correct. If not, you repeat the procedure from the beginning, without remaking the mistakes you made before.

Another example is a game for which you don't know the rules and little by little you understand them thanks to some attempts until the moment you know how to play it.

In any case, we see this interaction between a *Learner* and an entity (the vending machine or the game) called the *Teacher*. It is interesting to replicate the behavior of the *Learner* in our computers to simulate the learning process from the *Teacher*. The goal of this report is to focus on the $L*$ and the $NL*$ algorithms which are *Learning* algorithms of regular languages and have been proposed respectively by Dana Angluin's [1] and Benedikt Bollig *et al.* [4].

After introducing how these algorithms work, we will talk about the implementation and proceed with a complete comparison.

2 Regular Languages and Finite State Automata

In this section we will introduce the definitions of *Regular Language* and *Automaton* over an alphabet Σ since these notions are crucial to understand the two algorithms L* and NL*.

Theorem 1 (Kleene's theorem)**.** A language is called regular if it can be represented by a regular expression or equivalently by a finite state automaton.

Let Σ^* be the set of all words over the alphabet Σ (including the word of length 0 noted ε).
A recursive definition of a regular language can be expressed by the following grammar:

$$L = \varnothing \mid \varepsilon \mid \alpha \mid L \cup L \mid \overline{L} \mid L \cdot L \mid L^*$$

where:

- $\varnothing$ is the empty language;
- ε is the word of length 0;
- α is a symbol belonging to the alphabet Σ;
- $L_1 \cup L_2 = \{\omega \in \Sigma^* \mid \omega \in L_1 \vee \omega \in L_2\}$, which is the union of L_1 and L_2 ;
- $L_1 \cdot L_2 = \{\omega \in \Sigma \mid \exists w_1 \in L_1, \exists w_2 \in L_2 \text{ and } w_1 \cdot w_2 = \omega^1\}$, which is the concatenation of all words in L_1 with all words in L_2;
- $\overline{L} = \{\omega \in \Sigma^* \mid \omega \notin L\}$, called the complement of the language L;
- To define L^* let $L^0 = \varepsilon$, $L^1 = L$ and $L^{i+1} = L^i \cdot L$, then $L^* = \bigcup_{i=0}^{\infty} L_i$. This operation is called the *Kleene's star*

Some properties are not listed since they can be derived from the previous definition, for example the intersection betweem languages ($L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$).

Definition 2.1 (Automaton)**.** An automaton $\mathcal{A}$ is represented by a 5-tuple $\langle \Sigma, \mathcal{Q}, \delta, q_I, F \rangle$ where Σ is the alphabet, Q is the set of states composing the automaton, δ is the set of transition, $q_I \in Q$ is the initial state and $F \subseteq \mathcal{Q}$ is the set of accepting states. A transition δ is a mapping $Q \times \Sigma \to Q$.

Example 1. A transition like $\delta(q_i, \alpha) = q_j$ means that we can move from state q_i to state q_j when reading symbol α.

[1] Note $w_1 \cdot w_2$ is the concatenation of the word w_1 followed by the word w_2

A state q_i is called successor of q_j if it exists $\delta(q_i, \alpha) = q_j$ for some $\alpha \in \Sigma$. If q_j is successor of q_i then q_i is the predecessor of q_j.

Following the construction of the automaton $\mathcal{A}$, and especially focusing on its transition function, it is possible to classify $\mathcal{A}$ into two different classes.

Definition 2.2 (DFA and NFA)**.** If for every state $q_i \in Q$ and for every letter $\alpha \in \Sigma$ there exists only one successor by α then the automaton is *deterministic* and noted *DFA*, otherwise the automaton is called *non-deterministic* and noted *NFA*.

We can define a *DFA* also with a logic formula:

$$\forall q_i \in \mathcal{Q}, \forall \alpha \in \Sigma \text{ if } \exists \delta_1, \delta_2 \in \delta \text{ such that } \delta_1(q_i, \alpha) = q'_j \text{ and } \delta_2(q_i, \alpha) = q''_j \text{ then } q'_j = q''_j$$

Since every regular language can be represented by an automaton, be it deterministic or not, the operations like intersection, union and complementation are also available on automata. Among these properties, the most difficult to compute is the complementation when the automaton is non-deterministic as it asks first to make the *NFA* deterministic [2] which is a *PSPACE-complete* problem.

However, *NFA*s have a big advantage since they can be exponentially smaller than the corresponding *DFA* and this may be interesting mainly if we care about space resources.

Remark. *NFA* and *DFA* have the same expressive power meaning that

$$\bigcup NFA = \bigcup DFA = \bigcup RegularLanguages$$

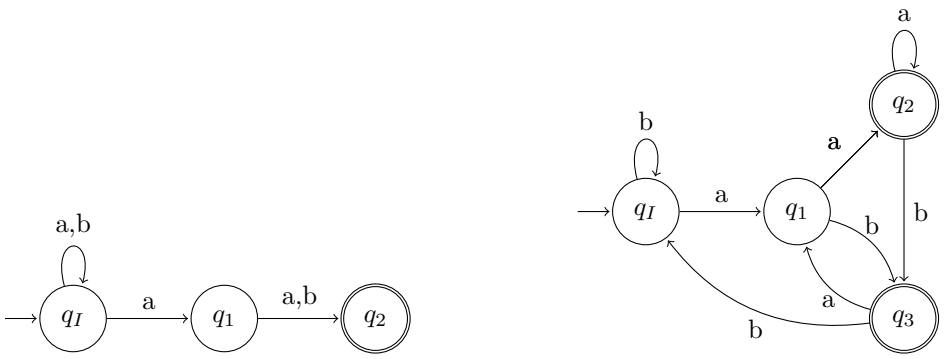
Definition 2.3 (Minimal automaton)**.** An automaton $\mathcal{A}$ is minimal if it is not possible to construct a smaller automaton $\mathcal{A}'$ for the same language having less states than $\mathcal{A}$. Moreover, a minimal *DFA* (noted *mDFA*), for a given regular language, is *unique*. However there may exists several minimal *NFA*, noted *mNFA*, for a regular language **U**.

Finally, we define, Regular Expressions (or *RegEx*) with the following grammar:

$$R = \omega \mid R + R \mid R \cdot R \mid R^* \mid (R)$$

where ω is a word belonging to Σ^*, the "+" symbol indicates the union operator, "$\cdot$" is the concatenation of two regular expression, the "$*$" is defined as for regular languages and finally parentheses allows to introduce priority over the other operators.

Example 2 (DFA vs NFA)**.** In Figure 1 we have an example of two minimal automata for the language $L = (a + b)^* a(a + b)$ where we can see that the NFA needs less states then the corresponding minimal DFA.



(a) NFA (b) DFA

Figure 1: Minimal NFA vs minimal DFA

2.1 Some notation

We will note an automaton $\mathcal{A}$, $L = \mathcal{L}(\mathcal{A})$ means that the regular language recognized by the automaton $\mathcal{A}$ is L. An expression of type $L = (a+b)$ indicates that L is recognized by the regular expression $(a+b)$. If not specified we suppose Σ to be equal to a, b and ω be a word over Σ^*.

[2] This operation is called determinisation

3 Before Learning

As said in Section 1, the goal of L^* and NL^* is to "learn" a regular unknown language $\mathbf{U}$ and output a conjecture which, when accepted, will be an automaton $\mathcal{A}$ recognizing $\mathbf{U}$.

Both algorithms work on the idea that a *Learner* interacts with a *Teacher* knowing $\mathbf{U}$ via two kinds of queries: *membership* and *equivalence* queries.

In a *membership query* the *Learner* asks the *Teacher* if a word ω belongs to $\mathbf{U}$.

$$member(\omega) = \begin{cases} \text{True} & \text{if } \omega \in \mathbf{U} \\ \text{False} & \text{otherwise} \end{cases} \tag{1}$$

In an *equivalence query* the *Learner* sends an Automaton (representing a conjecture) to the *Teacher* which can answer *Yes* if $\mathcal{L}(\mathcal{A}) = \mathbf{U}$, or a word ω belonging to $\mathbf{U}$ but not to $\mathcal{L}(\mathcal{A})$ or the other way round. In this case, ω is called a *counter-example*.

A counter-example can also be defined as a word ω belonging to the symmetrical difference (noted Δ) between $\mathcal{L}(\mathcal{A})$ and $\mathbf{U}$.

$$equiv(\mathcal{A}) = \begin{cases} \text{Yes} & \text{if } \mathcal{L}(\mathcal{A}) = \mathbf{U} \\ \omega \in \mathbf{U} \Delta \mathcal{L}(\mathcal{A}) & \text{otherwise} \end{cases} \tag{2}$$

4 Learning with L*

The L^* algorithm, described by Dana Angluin in [1], is a learning algorithm whose goal is to build a *mDFA* recognizing an unknown language $\mathbf{U}$ thanks to membership and equivalence queries. This operation can be done through some important bricks that are partially reused by the NL^* algorithm.

4.1 The observation table

In the previous section, we have talked about how the *Learner* and the *Teacher* interact, next we show how the *Learner* stores the received answers in its memory, called *observation table (O.T)*.

$T : \omega \to \{0, 1\}$ is a mapping allowing to know if the *Teacher* has accepted the word ω.

Definition 4.1 (Observation Table)**.** The observation table is a matrix of booleans whose rows are indexed by the elements of a non-empty set of words called S and whose columns are indexed by the elements of another non-empty set of words called E. And where the content of cell $c_{i,j}$ is $T(S[i] \cdot E[j])$[3].

S and E are respectively *prefix-closed* and *suffix-closed*.

Definition 4.2 (Prefix (resp. Suffix))**.** A prefix (resp. suffix) of a word ω, noted $Pref(\omega)$ (resp. $Suff(\omega)$), is a word $\omega_1 \in \Sigma^*$ such that $\exists \omega_2 \in \Sigma^*$ where $\omega_1 \cdot \omega_2 = \omega$ (resp. $\omega_2 \cdot \omega_1 = \omega$).

Definition 4.3 (Prefix-closedness and Suffix-closedness)**.** A non-empty set X of words is *prefix-closed* if

$$\forall \omega \in X, \forall Pref(\omega), Pref(\omega) \in X$$

and suffix-closed if

$$\forall \omega \in X, \forall Suff(\omega), Suff(\omega) \in X$$

In both cases, X should contain ε.

The *O.T.* must be completed by all words of the form $\omega \cdot \alpha$ for $\forall \omega \in S$ and $\forall \alpha \in \Sigma$ if $\omega \cdot \alpha \notin S$. This set of new words is added to the rows of the *O.T.* and stored in a set called SA.

$$SA = \{\forall s \in S, \forall c \in \Sigma : s \cdot c \mid s \cdot c \notin S\}$$

Let S_{ext} to be $S_{ext} = S \cup SA$.

Definition 4.4 (Completeness)**.** The *O.T.* is complete if $\forall s \in S, \forall c \in \Sigma, \exists s' \in \Sigma \mid s \cdot c = s'$

[3] $S[i]$ (resp. $E[j]$) indicates the i^{th} (resp j^{th}) element of S (resp. E). Even if sets are meant not to have the notion of order, we consider for simplicity that the i^{th} element of the set is the i^{th} element added in it

Example 3. If $S = \{\varepsilon, a, aa\}$ and $\Sigma = \{a, b\}$ then

$$SA = \{b, ab, aaa, aab\}$$ [4]
$$S_{ext} = \{\varepsilon, a, aa, b, ab, aaa, aab\}$$

Notation 1 (The row function)**.** For every ω in S_{ext}, $row(\omega)$ is the bit-vector made by the concatenation of all $T(\omega \cdot \omega')$ for $\omega' \in E$.

Remark. Adding a word ω in S_{ext} means that a membership query is sent for every word in $\{\forall e \in E \mid \omega \cdot e\}$. Similarly, adding a word ω' means that a membership query is sent for all words in $\{\forall s \in S_{ext} \mid s \cdot \omega'\}$.

4.2 Closedness and Consistency in L*

In both algorithms, the *O.T.* should respect the closedness and the consistency properties. Notice that their definitions vary for the two algorithms. In this section, we define these two properties for the L* algorithm.

Definition 4.5 (Closedness)**.** The *O.T.* is closed if

$$\forall \omega \in SA, \exists \omega' \in S : row(\omega) = row(\omega')$$

Definition 4.6 (Row Similarity)**.** Two rows r_1, r_2 are similar if they have the same corresponding bit-vector.

Definition 4.7 (Row Promotion)**.** Given a word $\omega \in SA$, we say that $row(\omega)$ is promoted when ω is moved from SA to S. After a row promotion, the *O.T.* must be completed again.

Lemma 1. If we move an element from SA to S then S remains *prefix-closed*.

Proof. Every element ω of SA is the concatenation of an element $\omega' \in S$ and a letter $\alpha \in \Sigma$. We have that

$$Pref(\omega) = Pref(\omega' \cdot \alpha) = \{\omega' \cdot \alpha\} \cup Pref(\omega')$$

By definition, S is prefix-closed, so if $\omega \in S$, then every prefix of ω' is already in S. Therefore adding ω in S will not impact the prefix-closedness of S. □

If the table is not closed then $\exists \omega \in SA$ such that $\forall \omega' \in S \mid row(\omega) \neq row(\omega')$. To make the *O.T.* closed we promote $row(\omega)$ and then, as said in Definition 4.7, we complete again the *O.T.*

Definition 4.8 (Consistency)**.** The *O.T.* is consistent if

$$\forall \omega_1, \omega_2 \in S, \forall \alpha \in \Sigma : row(\omega_1) = row(\omega_2) \rightarrow row(\omega_1 \cdot \alpha) = row(\omega_2 \cdot \alpha)$$

If the table is not consistent, then there exists $row(\omega_1) = row(\omega_2)$ and $row(\omega_1 \cdot \alpha) \neq row(\omega_2 \cdot \alpha)$ for some $\alpha \in \Sigma$ and some $\omega_1, \omega_2 \in S$. We can restore the consistency property by making ω_1 and ω_2 no more similar. To do this, we have to add a new column in the *O.T.* such that the new bit b_1 appended to ω_1 is different to the bit n_2 appended to ω_2. We look, therefore, for an $\alpha \in \Sigma$ and an $e \in E$ such that $T(\omega_1 \cdot \alpha \cdot e) \neq T(\omega_2 \cdot \alpha \cdot e)$ and we add $\alpha \cdot e$ in E. In this way we are sure that $b_1 = T(\omega_1 \cdot \alpha \cdot e)$ will be different from $b_2 = T(\omega_2 \cdot \alpha \cdot e)$.

We can now see the utility of the set SA. When we look for $T(\omega \cdot \alpha \cdot e)$ where $\omega \in S$, we are guaranteed that $\omega' = \omega \cdot \alpha$ exists in S_{ext} since the table is complete. Therefore, $T(\omega \cdot \alpha \cdot e) = T(\omega' \cdot e)$ exists.

Lemma 2. If we add a word $\omega = \alpha \cdot e$ where $\alpha \in \Sigma$ and $e \in E$ then E remains *suffix-closed*.

The proof is analogous to the one of Lemma 1.

[4] b is in SA since it is the concatenation of $\varepsilon \in S$ and $b \in \Sigma$

4.3 L*: Automaton creation

When the observation table is closed and consistent, the Learner can send to Teacher a conjecture of the language learnt so far.

The conjecture is an automaton where:

- $Q = \{row(\omega), \forall \omega \in S\}$;
- $\delta(row(\omega), \alpha) = row(\omega \cdot \alpha, \forall \omega \in S, \alpha \in \Sigma$[5]$)$;
- $q_I = row(\varepsilon)$;
- $F = \{row(\omega) \mid \omega \in S \text{ and } T(\omega) = 1\}$.

After sending the conjecture, if the Teacher answers *Yes* the algorithm stops, otherwise the counter-example ω and all of its prefixes are added to S. The *O.T.* is completed and the algorithm restart.

The following lemmas are a reformulation of those given in [1].

Lemma 3. Let q_I be the initial state and ω a word in S_{ext} then $\delta(q_I, \omega) = row(\omega)$.

Proof. We prove this by induction. Take a word $\omega \in S_{ext}$ and let q_I be the initial state. If $length(\omega) = 0$, then $\delta(q_I, \omega) = \delta(q_I, \varepsilon) = \delta(row(\varepsilon), \varepsilon) = row(\varepsilon \cdot \varepsilon) = row(\varepsilon) = q_I$ that is true by definition. Suppose this property is true for every word $\omega_n \in S_{ext}$ of length n, we want to prove that $\delta(q_I, \omega) = row(\omega)$ for a word $\omega \in S_{ext}$ of length $n+1$. Let ω be a word of length $n+1$ and let $\omega = \omega_n \cdot \alpha$ where $\alpha \in \Sigma$ and ω_n is the prefix of ω with length n. Since S_{ext} is prefix closed, then ω_n should exists. We have $\delta(q_I, \omega) = \delta(q_I, \omega_n \cdot \alpha) = \delta(\delta(q_I, \omega_n), \alpha)$. By inductive hypothesis, $\delta(q_I, \omega_n)$ exists since $length(\omega_n) = n$ and is equal to $row(\omega_n)$, therefore $\delta(\delta(q_I, \omega_n), \alpha) = \delta(row(\omega_n), \alpha)$. The *O.T.* is closed so $\exists s \in S$ such that $row(\omega_n) = row(s)$ and so $\delta(row(\omega_n), \alpha) = \delta(row(s), \alpha)$. By definition of the transition function, $\delta(row(s), \alpha) = row(s \cdot \alpha)$ and by construction of the $O.T$, for every $s \in S$ there exists an $s' \in S_{ext}$ such that $s' = s \cdot \alpha$. We can conclude that for every $\omega \in S_{ext}$ there exist a state represented by $row(\omega)$ such that $\delta(q_I, \omega) = row(\omega)$. □

Lemma 4. Let $\omega \in S_{ext}$ and $e \in E$, then $\delta(q_I, \omega \cdot e)$ leads to an accepting state if $T(\omega \cdot e) = 1$.

Proof. As previous lemma, we prove this by induction on the length of the word e. If $length(e) = 0$, then $e = \varepsilon$ and by definition, $row(\omega \cdot \varepsilon) = row(\omega)$ is acceptant if $T(\omega) = 1$. Suppose this property is true for every word e_n of length n. Let $e_{n+1} \in E$ be a word of length $n+1$, $e_n \in E$ the suffix of e_{n+1} of length n and $\alpha \in \Sigma$ such that $\alpha \cdot e_n = e_{n+1}$. Since E is suffix close, we know that e_n should exist in E. We have that $\delta(q_I, \omega \cdot e_{n+1}) = \delta(q_I, \omega \cdot \alpha \cdot e_n) = \delta(\delta(q_I, \omega), \alpha \cdot e_n) = \delta(row(\omega), \alpha \cdot e_n) = \delta(\delta(row(\omega), \alpha), e_n)$. Since the table is closed, $\exists \omega' \in S$ such that $row(\omega) = row(\omega')$, and so $\delta(row(\omega), \alpha) = \delta(row(\omega'), \alpha) = row(\omega' \cdot \alpha)$ thanks to the preceding lemma. Again, since the table is closed, $\exists \omega'' \in S$ such that $row(\omega' \cdot \alpha) = row(\omega'')$. We have now that $\delta(q_I, \omega \cdot e_{n+1}) = \delta(row(\omega''), e_n) = \delta(\delta(q_I, \omega''), e_n) = \delta(q_I, \omega'' \cdot e_n)$ and since the length of e is n then, by inductive hypothesis, $\delta(q_I, \omega'' \cdot e_n)$ leads to an accepting state if $T(\omega'' \cdot e_n) = 1$. We can conclude that $\delta(q_I, \omega \cdot e_{n+1})$ leads in an accepting state if $T(\omega \cdot e_{n+1}) = 1$. □

Lemma 5. The automaton created is deterministic.

Proof. We know that for every row $s_1 \in S$ and for every $\alpha \in \Sigma$, $s_1 \cdot \alpha \in S_{ext}$. Moreover, since the table is consistent, for every $s_2 \in S$, if $row(s_1)$ and $row(s_2)$ are similar, we have that $row(s_1 \cdot \alpha) = row(s_2 \cdot \alpha)$ for $\forall \alpha \in \Sigma$. Therefore there exists only one successor for every state $q \in Q$ and every letter $\alpha \in \Sigma$. □

Lemma 6. The automaton created is minimal.

Proof. We proceed by contradiction. Suppose that $\mathcal{A}$ is not minimal, then it is possible to merge[6] at least two states q_i, q_j to reduce the number of states. By construction, q_i and q_j are linked to two different rows r_i, r_j of the observation table. Let $\omega_1 \in S, \omega_2 \in S \mid row(\omega_1) = r_1$ and $row(\omega_2) = r_2$. Since $r_1 \neq r_2$ then $\exists e \in E \mid T(\omega_1 \cdot e) \neq T(\omega_2 \cdot e)$. Let $\mathcal{A}_1$ be the automaton $\mathcal{A}$ where $row(\omega_1)$ is the initial state and $\mathcal{A}_2$ be the automaton $\mathcal{A}$ where $row(\omega_2)$ is the initial state. The word e will only be accepted by one of the two automata, so since they do not recognize the same language, we can't merge them in $\mathcal{A}$. This contradict the initial hypothesis and so $\mathcal{A}$ is minimum. □

[5] Again we know that $\omega' = \omega \cdot \alpha$ exists in S_{ext} since the table is complete

[6] Two states can be merged if they recognize the same language or, because of the automaton is deterministic, they have same outgoing arrows

4.4 L* analysis

4.4.1 Correcteness

The goal of the algorithm is to find an automaton recognizing the language of the Teacher. The final conjecture found by the Learner is correct since it has been accepted by the Teacher.

4.4.2 Halting

The algorithm terminates only if the Teacher answers Yes to an *equivalence query* of the Learner, but is the Learner able to find this final *equivalence query* ?

We know, by definition, that the unknown language **U** is regular. It means that there exists an automaton $\mathcal{A}$ recognizing it with a finite number n of states.

If in a certain moment we have $n' < n$ different rows and the $O.T.$ is closed and consistent, the Learner make its conjecture. This conjecture cannot be the good one, since, from Lemma 6, the automaton $\mathcal{A}'$ of the conjecture cannot have less states then $\mathcal{A}$. The Teacher must returns a counter-example allowing to separate two similar rows. The Learner repeats the algorithm, and after every *equivalence query*, n' will increase by one. When $n' = n$ the conjecture will have n states, the same of $\mathcal{A}$, and therefore, the Teacher will answer Yes to the last conjecture allowing the algorithm to stop.

Remark. Note that Lemma 3 and Lemma 4 say that the automaton $\mathcal{A}$ created from a closed and consistent table recognizes correctly the language understood so far and since the *mDFA* is unique (see Definition 2.3) then the conjecture is the correct one.

4.4.3 Time complexity

Suppose that the L* algorithm (implemented in Algorithm 3) takes constant time to make the conjecture, to complete the $O.T.$, to promote rows and to check if the table is consistent and closed.

Then time complexity should be measured on the size of the $O.T.$ or equivalently on the number of cell composing it.

In the worst case, the number of columns n (that is the cardinal of E) equals the number of states $\mathcal{A}$.

The size of S depends on the length m of the counter-example provided by the Teacher. At least S has n rows, one per number of columns, but considering the length of m then $|S|$[7] will have $O(n \times m)$ rows.

The size of SA is at most $O(|\Sigma| \times |S|)$ since for every word in $|S|$ we can potentially have $|\Sigma|$ successors to add in SA.

In conclusion we have a matrix with a number of cells equal to

$$\begin{aligned} |E| \times (|S| + |SA|) &= O(|E| \times (|S| + \Sigma \times |S|)) \\ &= O(2 \cdot |\Sigma| \times |E| \times |S|) \\ &= O(n^2 \times m) \end{aligned}$$

Hence, the time complexity is polynomial on the size of the automaton and the larger counter-example given by the Teacher.

5 Learning with NL*

The NL* algorithm, described by Benedikt Bollig *et al.* in [4], is an algorithm based on L* whose goal is to learn an unknown language **U** building potentially smaller automata then the one built by L*. To achieve this goal, NL* outputs a particular type of automaton called *Canonical residual final state automata (cRFSA)*.

5.1 Residual final state automaton (RFSA)

Theorem 2. Every regular language L has a finite number of residuals.

Definition 5.1. A residual of a language L is a new language L' where $L = \{\omega \cdot v,$ for some $\omega \in \Sigma^*$ and for $\forall v \in L'\}$. This is commonly noted $\omega^{-1}L = L'$ for some $\omega \in \Sigma^*$.

[7] The cardinal of S

Example 4. Let $L = (aa)^*$ and $\Sigma = \{a\}$. L has only two residuals $L_1 = (aa)^*$ and $L_2 = a(aa)^*$. We can see that there does not exist any other residual because if we pose $E_{ven} = \omega \in \Sigma^* \mid len(\omega) \equiv 0 \mod 2$ and $O_{dd} = \omega \in \Sigma^* \mid len(\omega) \equiv 1 \mod 2$, we have $E_{ven} \cup O_{dd} = \Sigma^*$ and $E_{eve}^{-1}L = L_1$ and $O_{dd}^{-1}L = L_2$. No other residual is possible since there exists no other word in $\Sigma^*/\{E_{ven} \cup O_{dd}\} = \varnothing$ which can create a new residual.

Remark. The set of word $\mathcal{W}$ over an alphabet Σ is a syntactic monoid if for every $\omega \in \mathcal{W}$, $\omega^{-1}L$ gives the same residual. Moreover, each residual has a corresponding syntactic monoid.

Example 5. Turning back to Example 4, we can say that E_{ven} is the syntactic monoid generating $\mathcal{R}_\infty = (aa)^*$ and O_{dd} it the other syntactic monoid generating $\mathcal{R}_\in = a(aa)^*$.

Theorem 3 (mDFA and Residuals)**.** A DFA $\mathcal{A}$ is minimum if and only if, for a given language L, $\mathcal{L}(\mathcal{A}) = L$ and the number of states of A equals the number of the residual of L.

Proof. $\rightarrow$ Let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \delta, q_i, F \rangle$ be a *mDFA* and $\mathcal{A}_i$ the automaton $\mathcal{A}$ where q_I is a state $q_i \in \mathcal{Q}$. For every $q_i \in \mathcal{Q}$, $\mathcal{L}(\mathcal{A}_i)$ is a residual $\mathcal{R}_i$ of L. We can also define its syntactic monoid M by all words ω accepted by $\mathcal{A}$ having $F = q_i$.

$\leftarrow$ Given the set of residuals $\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n\}$ of L and an alphabet Σ. For every $\mathcal{R}_i$ there exists an associated monoid M_i such that $\omega^{-1}\mathcal{R}_i = L$ for all $\omega \in M_i$. We can construct the minimal DFA $\langle\ \Sigma,\ Q,\ \delta,\ q_I,\ F\ \rangle$ where:

- $\mathcal{Q} = \mathcal{R}$;
- $\delta = \{(\mathcal{R}_i, \alpha) = \mathcal{R}_j$ for every $\mathcal{R}_i, \mathcal{R}_j \in \mathcal{R}$ and $\alpha \in \Sigma$ if $\alpha^{-1}L(\mathcal{R}_i) = L(\mathcal{R}_j)\}$;
- q_I is the residual R_i such that $R_i = L$ (it is the residual whose syntactic monoid contains ε);
- $F = \{\mathcal{R}_i \in \mathcal{R}$ such that $\varepsilon \in \mathcal{R}_i\}$.

The created automaton is minimum since for any state $q_i, q_j, \mathcal{L}(\mathcal{A}_i) \neq \mathcal{L}(\mathcal{A}_j)$. □

Starting from these definitions, we can see that in L* every row is associated to a state of a mDFA and therefore, since every state is associated to a residual, we can deduce by transitivity that every row of the *O.T.* represents a residual of the language.

Definition 5.2. An *RFSA* is a *non-deterministic* automaton where every state represents a residual of L.

Definition 5.3 (Prime Residual)**.** Let $\mathcal{R}$ be the set of residuals a language L, then a residual $\mathcal{R}_i \in \mathcal{R}$ is *composed* if it exists a subset $\mathcal{R}' \subseteq \{\mathcal{R} - \{\mathcal{R}_i\}\}$ such that $R_i = \bigcup_{r \in \mathcal{R}'} r$. A residual which is not composed is called *prime*.

Definition 5.4 (Language Covering)**.** A language L' is covered by an other language L if $L' \subseteq L$ or equivalently $\forall \omega \in \Sigma^*, \omega \in L' \rightarrow \omega \in L'$

Definition 5.5. A *RFSA* is *canonical* if every state is associated to a *prime* residual.

Let $\mathcal{R}$ be the set of residual of L. A *canonical RFSA (cRFSA)* $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \delta, Q_i, F \rangle$ can be constructed as follow:

- $\mathcal{Q} = \{\mathcal{R}_i \subseteq \mathcal{R} \mid \mathcal{R}_i$ is prime$\}$;
- $\delta = \{(\mathcal{R}_i, \alpha) = \mathcal{R}_j . \forall \mathcal{R}_i, \mathcal{R}_j \in \mathcal{R}$ and $\forall \alpha \in \Sigma$ if $L(\mathcal{R}_i) \subseteq \alpha^{-1}L(\mathcal{R}_j)\}$;
- $Q_I = \{\mathcal{R}_i \in \mathcal{R} \mid \mathcal{R}_i \subseteq L\}$[8];
- $F = \{\mathcal{R}_i \in \mathcal{R} \mid \varepsilon \in \mathcal{R}_i\}$.

Theorem 4. A *cRFSA* can be exponentially smaller then a *mDFA*.

[7] A procedure to find residual of a language is proposed in Algorithm 5

[8] The initial state may not be unique, it is denoted an upper-case $\mathcal{Q}$

5.2 Closedness and Consistency in NL*

The NL* algorithm has similar properties as L* since the $O.T.$, the sets S, E, SA and S_{ext} and the mapping T are defined as in L*. However, the closedness and consistency properties are defined in a different way.

Definition 5.6 (Row Covering)**.** A row r_1 of the $O.T.$ is covered ($\sqsubseteq$) by a row r_2 if every bit of r_1 is less or equal of the corresponding bit in r_2.

Example 6. Given $r_i = [0, 1, 1, 0, 1]$ and $r_j = [0, 1, 1, 1, 1]$, we have that $r_i \sqsubseteq r_j$.

Definition 5.7 (Row Union)**.** The union ($\sqcup$) of two rows r_1, r_2 gives a new row r_3 where each bit b_i is obtained as the max of the i^{th} bit of r_1 and the i^{th} bit of r_2

Example 7. Given $r_i = [0, 1, 0, 0, 1]$ and $r_j = [0, 1, 0, 1, 1]$, we have that $r_i \sqcup r_j = [0, 1, 1, 1, 1]$.

Definition 5.8 (Composed and Prime Rows)**.** A row r of the $O.T.$ is composed if it exists a subset of rows R' not including r such that $r = \bigsqcup_{i \in R'} i$

Notation 2. From now on the function $Prime : \omega \to True, False$ is a function returning $True$ if $row(\omega)$ is Prime, $False$ otherwise.

Definition 5.9 (Closedness)**.** The $O.T.$ is closed if

$$\forall \omega \in SA, Prime(\omega) \to \exists \omega' \in S : row(\omega) = row(\omega')$$

To make the $O.T.$ close, the row ω is promoted and then the $O.T.$ is completed.

Definition 5.10 (Consistency)**.** The $O.T.$ is consistent if

$$\forall \omega_1, \omega_2 \in S, row(\omega_1) \sqsubseteq row(\omega_2) \to \forall \alpha \in \Sigma, row(\omega_1 \cdot \alpha) \sqsubseteq row(\omega_2 \cdot \alpha)$$

If the table is not consistent, $\exists \omega_1, \omega_2 \in S$ such that $row(\omega_1) \sqsubseteq row(\omega_2)$ and $row(\omega_1 \cdot \alpha) \not\sqsubseteq row(\omega_2 \cdot \alpha)$ for an $\alpha \in \Sigma$. We can restore the consistency by making ω_1 and ω_2 no more RFSA-similar: we find an $e \in E$ such that $T(\omega_1 \cdot \alpha \cdot e) \sqsupset T(\omega_2 \cdot \alpha \cdot e)$ and we add $\alpha \cdot e$ in E.

5.3 NL*: Automaton creation

When the $O.T.$ is closed and consistent, the Learner can send to the Teacher a conjecture of the language that it has understood so far.

The conjecture is the automaton where:

- $Q = \{row(\omega), \forall \omega \in S \text{ and } Prime(\omega)\}$;
- $\delta = \{\delta(row(\omega), \alpha) = row(\omega'), \forall \omega \in S, \alpha \in \Sigma, \omega' \in S_{ext} \mid Prime(\omega) \text{ and } row(\omega') \sqsubseteq row(\omega \cdot \alpha)\}$;
- $\mathcal{Q}_I = row(\omega), \forall \omega \in S \mid Prime(\omega) \text{ and } row(\omega) \sqsubseteq row(\varepsilon)$;
- $F = \{row(\omega) \mid \omega \in S \text{ and } T(\omega) = 1\}$.

After sending of the conjecture, if the Teacher answers *Yes* the algorithm stops, otherwise a counter-example ω is provided. ω is added to the E set and if needed E is completed to remain suffix-closed.

Remark. Notice that L* adds the counter-example in S while NL* adds it in E.

Lemma 7. Let $q_I \in Q_I$ be an initial state of $\mathcal{A}$ and $\omega \in S_{ext}$ then $\delta(q_I, \omega) \sqsubseteq row(\omega)$.

The proof is similar to the one of Lemma 3 considering that now we do not use the equality relation between states, but the covering ($\sqsubseteq$) one defined in Definition 5.6.

The word acceptance of $NL*$ can be proved similarly to Lemma 4.

We can also note that:

- the automaton created is no necessarily deterministic, since the transition function allows multiple successors for a given state and a given symbol;
- the automaton may have multiple initial states: all states whose row is covered by $row(\varepsilon)$;
- $row(\varepsilon)$ may not be in Q_I if it is a composed row;
- the automaton created is the *cRFSA* since the represented states are only those corresponding to a prime row.

5.4 NL* analysis

5.4.1 Correcteness

As for the L* algorithm, NL* is correct since the solution must be approved by the Teacher before stopping.

5.4.2 Halting

As for the L* algorithm, NL* terminates because the number of residual of the language is finite and if the conjecture is not accepted, the counter-example, will allow the Learner to find a new residual until all *prime* residuals are found.

A little precision of why NL* adds the counter-example ω to E instead of S. In general, the given counter-example belongs to a residual that is not correctly represented by the sent automaton. However, while constructing the *O.T.* we only have partial information on the residuals, because $|E|$ may not be big enough to distinguish the between residuals. It may happen that the partial row associated to ω is not prime and therefore it will not be considered by the algorithm when testing the consistency and the closedness. On the other hand, adding ω into E is a correct way find directly new information on residuals. This allows NL* to terminate and, therefore, to find the correct conjecture.

5.4.3 Time complexity

In the worst case, the *cRFSA* has the same number of state of the *mDFA* and therefore the *O.T.* will have the same size of the *O.T.* of L*. Again the complexity depends on the length of the Teacher's counter-example. At the end we can conclude that the time complexity is in the worst case $O(n^2 \times m)$.

However since the *cRFSA* found by NL* may be smaller then the *mDFA* it happens that NL* finds the solution more quickly then L* and therefore the time needed to understand **U** can be smaller.

6 My implementation of the L* and NL*

The two algorithms have been implemented in *TypeScript*, the typed version of *Javascript* which has the advantage that, once compiled, can be run both on browsers and terminals thanks to *NodeJs* (a good *Javascript* run time environment).

The *GitHub* repository https://github.com/FissoreD/TER-M1-S2 is organized as follows:

- *pdf_src* contains the main papers I studied during the *TER*;
- *report* with the LaTeX source files to build this report;
- *src* containing the *TypeScript* source files;
- *public* where I put the public *JavaScript* libraries to render *Automata* (the *d3-graphviz.js* file) and to make operations between *Automata* (the *noam.js* file);
- *dist* which is the folder where the *TypeScript* compiler creates the compiled *JavaScript* files;
- *statistics* containing the results of my comparisons of performances between *L** and *NL**;

6.1 My Teacher

A major problem that I soon had to overcome was how to represent the interactions between the Learner and the Teacher, (see Equation (1)). That's why I did not only have to implement the Learner, but also the Teacher.

As said in the Introduction, a *Teacher* is an oracle knowing the language and answering to membership and equivalence queries. It is quite easy to see how a membership query can be answered: the Teacher can test if the received word belongs to its language or not.

However it is less evident how the *Teacher* could verify if a conjecture is valid or not and how to send the counter-example.

Definition 6.1. In [1], a *Teacher* providing every time a counter-example of shortest length is called *Minimal Adequate Teacher*.

Following this idea, I have decided to implement this kind of Teacher because, since the complexity of the algorithms also depends on the length of the counter-examples, it would have been more interesting to reduce as much as possible the value of m obtaining therefore a complexity of $O(n^3)$.

From this moment on, I will consider Teachers to be *Minimal Adequate Teachers*.

When the *Learner* sends the conjecture $\mathcal{A}$, the *Teacher* has to perform the symmetrical difference between $\mathbf{U}$ and $\mathcal{L}(\mathcal{A})$ to find the counter-example of shortest length. This operation can be made in the following way:

Algorithm 1: Shortest counter-example in $U \Delta \mathcal{L}(\mathcal{A})$

1 $\mathcal{A}_1 \leftarrow \mathcal{L}(\mathcal{A}) \cap \overline{\mathbf{U}}$;
2 $\mathcal{A}_2 \leftarrow \overline{\mathcal{L}(\mathcal{A})} \cap \mathbf{U}$;
3 $w_1 \leftarrow$ the shortest word accepted by $\mathcal{A}_1$;
4 $w_2 \leftarrow$ the shortest word accepted by $\mathcal{A}_2$;
5 **return** shortest(w_1, w_2);

The first two lines of this code are in fact those who are the most expensive in term of complexity in my project.

But first, before executing this algorithm, I had to find a way to correctly represents the Learner's conjecture and the Teacher's language to operate in a second moment the intersection and the complementation.

6.1.1 Learner's Conjecture and Teacher's Language representation

The best way I found to represent the Teacher's Language was to think that the Learners return an Automaton when providing their conjecture. That's why I decided to also represent the Teacher's Language in the form of an automaton.

I implemented the Automaton class which has some key attributes to identify the main characteristics of an automaton on top of this I have also created a *State* class representing the states of the *Automaton* where I added the information of successors (and predecessors) as attributes attached to the instancies of the class.

```
class State {
  isAccepting: boolean;
  isInitial: boolean;
  alphabet: string[];
  outTransitions: Map<string, State[]>;
  inTransitions: Map<string, State[]>;
  successors: Set<State>;
  predecessor: Set<State>;
  name: string;

  ...

  addTransition(symbol: string, state: State): void;
  getSuccessor(symbol: string): State;
  getPredecessor(symbol: string): State;
}
```

This implementation exploits the *Map* data-structure which is a fast way in *JavaScript* to get a value from a given key. Since the Automaton can be non-deterministic, *outTransition* associates a list of *States* to every symbol of the alphabet.

Moreover a *State* "knows" if it is initial and accepting.

Now that I had the representation of a state, the class Automaton could be represented as:

```
class Automaton {
  states: Map<string, State>;
  initialStates: State[];
  acceptingStates: State[];
  allStates: State[];
```

```
  alphabet: string[];

  ...

  accept_word_nfa(word: string): boolean;
  minimize(): Automaton;
}
```

We can see that all the fundamental elements of the automaton are represented, there is the list of states, followed by the list of accepting and initial states. The alphabet is a list of string.

The transition function δ is a bit more difficult to see. But it can be deduced thanks to the *outTransition* attributes of a *State*.

After the *State* and the *Automaton* class, I needed a way to make the union, intersection, and complementation of two automata.

To do that I used the *noam.js* library [6]. The next thing that I had to do was therefore to create an *Adapter* to transform my representation of an *Automaton* to its representation and vice-versa.

Having these ingredients, $\mathcal{L}(\mathcal{A}) \cap \overline{\mathbf{U}}$ and $\overline{\mathcal{L}(\mathcal{A})} \cap \mathbf{U}$ could be easily calculated. Notice that the complementation of an automaton takes linear time if the automaton is deterministic (i.e. we modify the final states in $F' = \mathcal{Q} - F$) but if the automaton is non-deterministic, that's the case for the *NL** algorithm, the only way we know to make the complementation is to first determinize it which is a computationally difficult task.

6.1.2 Find the shortest counter-example

To find the *shortest* counter-example, I perform a *BFS* search from the initial states[9] to find the word of shortest length accepted by the Automaton.

I want to underline that if two automata, $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent then $\mathcal{A}_1 \Delta \mathcal{A}_2$ gives a new automaton $\mathcal{A}_3$ which recognizes the emptyset $\varnothing$. If $\mathcal{A}_1 \neq \mathcal{A}_2$ then it exists at least a word which is accepted by $\mathcal{A}_1$ and not by $\mathcal{A}_2$ or vice-versa. When I launch the *BFS* in $\mathcal{A}_3$, if no word is found, I return $\varnothing$ (I symbolize it with a string of length 0 in *JavaScript*) or the counter-example in the other case.

6.2 The Observation Table

The *O.T.* is implemented like this.

```
class Observation_table {
  private rows: string[];
  private columns: string[];
  private matrix: boolean[][];

  ...

  add_column(column_name: string): void;
  set_value(row_name: string, column_name: string, bool: boolean): void;
  add_row(row_name: string): void;
  get_value(row_name: string, column_name: string): boolean;
  get_row(row_name: string): boolean[];
}
```

It is a bidimensional matrix of booleans indexed by rows (the S and SA sets) and columns (the E set).

If the Learner wants to set to b to value of the cell corresponding to the concatenation of the words ω_1 and ω_2 then

$$matrix[rows.indexOf(\omega_1)][columns.indexOf(\omega_2)] = b;$$

Notice that the *matrix* is initialized with $matrix = [[]]$. When we add an element e at position (x, y) with $x > len(matrix)$ and $y > len(matrix[0])$, *JavaScript* won't raise the error *OutOfBound* as other programming languages, but instead it will automatically allocate the needed memory and affect the $matrix[x][y]$ to e.

[9] The initial states can be multiple in the NL*, that's why a use the plural.

6.3 The Learners implementation

In this section I describe the representation of the two Learners.

First of all, I realized that L* and NL* have in common some definitions. That's why I created a common abstract class which implements the shared methods and attributes of the algorithms.

```
1  abstract class LearnerBase {
2    alphabet: string[];
3    S: string[];
4    E: string[];
5    observation_table: ObservationTable;
6    SA: string[];
7    teacher: Teacher;
8
9    ...
10
11   update_observation_table(key: string, value: string): void;
12   make_member(pref: string, suff: string): string;
13   make_equiv(a: Automaton): string;
14   add_elt_in_S(new_elt: string): void;
15   add_elt_in_E(new_elt: string): void;
16   move_from_SA_to_S(elt: string): void;
17
18   abstract table_to_update_after_equiv(answer: string): void;
19   abstract make_automaton(): Automaton;
20   abstract is_close(): string | undefined;
21   abstract is_consistent(): string[] | undefined;
22 }
```

The two methods *make_member* and *make_equiv* reproduce the interaction with the Teacher. In the two cases the Learner returns a string that is:

- "1" or "0" for membership queries, if the word is accepted or not by the Teacher[10];
- a word ω which represents the counter-example. If $len(\omega) = 0$ then the Learner has understood the language.

add_elt_in_S and *add_elt_in_E* modify the content of the *O.T.* while adding a row to S as well as all of its prefixes or a column with all its suffixes, respectively.

The method *move_from_SA_to_S* has the same behavior as the *add_elt_in_S* plus it removes the row moved from SA.

We can see that the abstract methods [Line 20 to 23] correspond to the closedness and consistency properties and to the automaton creation. I also want to mention the *table_to_update_after_equiv* function which modifies the set S if the Learner is L*, the set E otherwise. Both L* and NL* have a dedicated class extending the *LearnerBase* class.

The NL* algorithm has in addition the *prime_lines* attribute storing the list of prime rows. It is constantly updated when the *O.T.* is modified thanks to some useful methods:

```
1  class NL_start extends LearnerBase {
2    prime_rows: string[];
3
4    ...
5
6    is_prime(row_key: string): boolean;
7    row_union(row1: string, row2: string): string;
8    is_covered(row1: string, row2: string): boolean;
9    check_prime_lines(): void;
10 }
```

These methods are implemented following the definition given in Definition 5.6 and Definition 5.7.

[10] In Javascript the type *char* does not exist

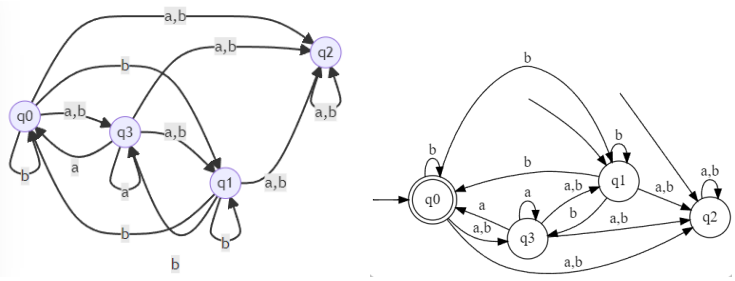
6.4 The HTML page

All these classes are meant to implement the essential bricks to make the learning algorithms work. However I wanted to display these interactions graphically via *HTML*.

To do so, I have created some dedicated classes to manage the web page and display the *O.T.* , the construction of the automata together with the message linked to the closedness and consistency properties.

At first I have considered the *Mermaid* library [7]. The automata displayed were nice, but when they started to have a huge number of states and transitions the drawing started to become less intelligible.

After some further research, I found a way to represent *Dot* files with the *Graphviz* library [8], that was more flexible.



(a) Automaton in Mermaid (b) Automaton with Graphviz

Figure 2: Mermaid vs Graphviz

In Figure 2 we see an example of the two automata representation. The initial and final states are not clearly indicated in the *Mermaid* plot and the transitions are less clear to understand.

Once I have finished to implement all of this, I have been able to publish my *HTML* page at the url https://fissored.github.io/TER-M1-S2/.

This page allows you to choose to run both the L^* and the NL^* algorithms. Some preloaded Teachers are proposed, but it is also possible to create a custom Teacher by entering a *Regular Expression*.

The regular expression should be encoded with the following grammar:

$$R = symbol \mid R + R \mid RR \mid R^* \mid (R)$$

The alphabet is composed by every alpha-numeric character and the empty string ε is encoded with \$.

Example 8.

$$(a + b) + \$ + ab^* + (ab)^*$$

indicates the language containing $\{\varepsilon, a, b, ab, abb, abbb \dots, ab, abab, ababab \dots\}$.

Once the Teacher created, the user can run step by step or the entire algorithm and every phase of the execution will be displayed.

In the *HTML* page, the *Message* section contains information about the *closedness* and the *consistency* of the $O.T$. Once these two properties are satisfied, the automaton corresponding to the conjecture is displayed (an example is given in Figure 2b).

When an automaton is displayed, the user is able to enter a word in the corresponding input section and test if it is accepted or not by the language recognized by the automaton. It can be useful for example to see immediately if the counter-example proposed by the Teacher is accepted or not by the automaton.

The history of the operations made by the *Learner* is saved in memory and can be re-viewed by clicking on the arrows on the side of the screen.

7 The comparison

Once compiled in JavaScript it is possible to run both algorithms, with *NodeJs* in the terminal. We have therefore exploited this possibility to launch the algorithms over some chosen *Regular Expression* to test the Learners' performances. A comparison between the two algorithms is proposed in paper [4]: Bollig *et al.* count the number of states of the final automata given by the Learners and accepted by

[10] The counter-example is also written in the *Message* section.

the Teacher along with the number of membership and equivalence queries. We added some personal criteria of comparison in order to also see how many time the *O.T.* is found not close or not consistent (We will call them closedness and consistency problems) and the number of transitions of the automata.

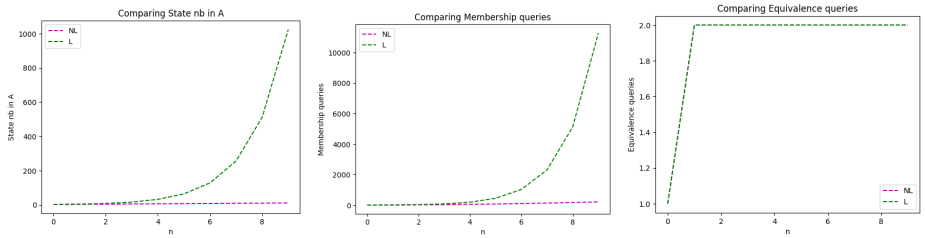
All of these statistics can be obtained by executing the files into the *test_nodejs* folder. The script creates a CSV file in which all the comparison values are stored and then a *Python* file can parse the CSV to finally transform the resulting CSV into plots.

The statistics aim to see the strength and the weaknesses of the to Learners, therefore we have tested them over specific *Regular Expressions (RegEx)*. These *RegEx* take into account the size of the *cRFSA*: since it *can* be exponentially smaller than the size of the *mDFA*.

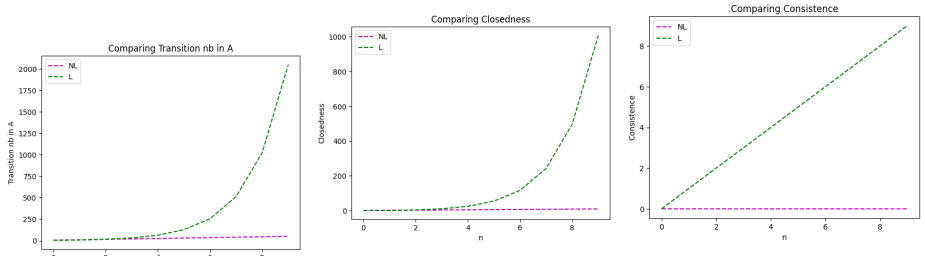
7.1 The cRFSA is exponentially smaller then the mDFA

Languages recognized by the *RegEx* $\mathbf{U} = (a+b)^*a(a+b)^n$ for a fixed n are known to build *mDFA* whose number of states increase exponentially every time we increase n. It is also known that these same *RegEx* can be represented by *non-deterministic* automata (*NFA*) whose number of states grows linearly with n.

If we calculate the *prime* residuals of $\mathbf{U}$ we can see that their number equals to $n+1$. We can intuitively understand that $\mathbf{U}$ can be represented by a *cRFSA* with the same number of states as the *NFA*.



(a) Comparing State Number (b) Membership queries Number (c) Equivalence queries Number



(d) Transition Number (e) Closedness Problem Number (f) Consistency Problem Number

Figure 3: mDFA vs cRFSA on $U = (a+b)^*a(a+b)^n$)

As expected the number of states of the *mDFA*'s curve is exponential compared to n and this also impacts the number of transitions, membership queries and closedness problems.

In Figure 3f, we see that the number of Consistency Problems grows linearly for L^* and it is close to zero for NL^*.

Let's analyze at first L*.

Let $\mathbf{U} = (a+b)^*a(a+b)^n$. L^* sends the three membership queries for ε, a and b. After that the table is closed and consistent and the first conjecture is an automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \varnothing$. This is not the good automaton and the Teacher gives back a counter-example $\omega = a^{n+1}$ of the shortest length[11]. All prefixes p of ω are added in $\mathbf{U}$ and $p \notin \mathbf{U}$. We have that the table is not consistent: $row(\varepsilon) = row(a^n)$ but $row(\varepsilon \cdot a) \neq row(a^n \cdot a)$. So L^* adds the new column a. This new column will make $row(\varepsilon) = row(a^{n-1})$ but $row(\varepsilon \cdot a) \neq row(a^{n-1} \cdot a)$. We continue in this way n times until $E = \{\varepsilon, a, \ldots, a^n\}$. This will stop the table to be not consistent since there will be no more similar rows in S. Since the automaton associated to $\mathbf{U}$ has precisely 2^{n+1} states therefore S will have to contain 2^{n+1} different rows. We know that after the first counter-example ω, $|S| = n+2$[12] will have to find $2^{n+1} - (n+1)$ closedness problems to promote the necessary number of rows from SA to S.

[10] To display plots we used the *matplotlib* library [3]

[11] The Teacher is supposed to be the *Minimal Adequate Teacher*

[12] ε plus the $n+1$ prefixes of ω

Let's now analyze the NL* behavior.

This algorithm, as said in Section 5, tries to find all prime residuals of $\mathbf{U}$. After the first conjecture, that, as for the L* is supposed to accept the language $\mathbf{U} = \varepsilon$, the Learner receives the counter-example $\omega = a^n$. Since *NL* adds the counter-example to E and so it will directly make $E = \{\varepsilon, a, \dots, a^n\}$. This coincides with the number of residuals of $\mathbf{U}$. In this case the algorithms only has to promote n rows, one for each residual and this is possible since the promotion of a rows will create new rows in SA to make the *O.T.* complete.

We can finally note that the number of equivalence queries is the same for the two algorithms since:

- if $n = 0$, then $\mathbf{U} = (a + b)^* a$ and the first equivalence query is immediately accepted, without any consistency problem;
- else: the two Learners only need two equivalence queries to understand $\mathbf{U}$[13]. After the first equivalence query, L* and NL* receive the first counter-example and thanks to the consistency and closedness check, both algorithms will be able to send a second conjecture $\mathcal{A}$ where $\mathcal{L}(\mathcal{A}) = \mathbf{U}$.

We can conclude that when the *cRFSA* is exponentially smaller than the *mDFA* it is better to apply the *NL** algorithm.

7.2 The cRFSA has the same size as the mDFA

In this section we are going to show the behavior of the two algorithms on a particular class of regular expressions depending on a fixed parameter n where the *cRFSA* has exactly the same size of the *mDFA* but where the corresponding minimum *NFA* is exponentially smaller. This automaton is proposed in Section 6 of [2]. The construction is done for an automaton $A_n = \langle \Sigma, Q, Q_I, F, \delta \rangle$ where:

- $\Sigma = a, b,$
- $Q = \{q_i \mid 0 \leq i < n - 1\},$
- $Q_I = \{q_i \mid 0 \leq i < n/2\},$
- $F = q_0$
- $\delta(q_i, a) = q_i + 1$for $0 \leq i < n - 1$, $\delta(q_{n-1}, a) = q_0$, $\delta(q_0, b) = q_0$, $\delta(q_i, b) = q_{i-1}$ for $1 < i < n$ and $\delta(q_1, b) = q_{n-1}$.

As proved in that paper, the number of states of a minimal *NFA* equals n, but the number of states of the *cRFSA* is exponential with respect to n. The statistics of the execution of the two algorithms are shown in Figure 4.

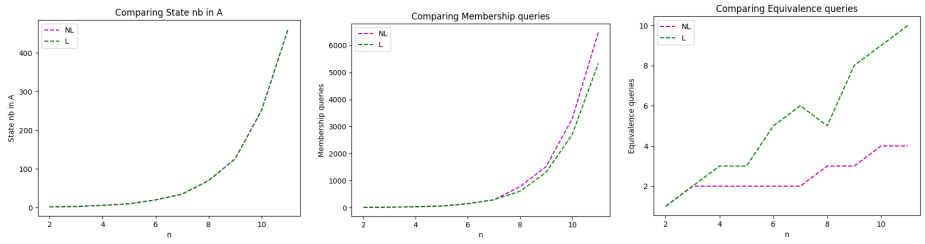
We can see that the number of state of the two automata generated by *NL** and *L** are the same and that the number of closedness problems and membership queries are similar. What changes, is the number of equivalence queries which is particularly smaller for the *NL** algorithm as for the number of time where the *O.T.* is found not consistent. This is because adding counter-examples in E instead of S is in general a good mean to be able to distinguish faster newer residuals. We are able to reduce therefore the number of Consistency problems and so the number of Equivalence queries.

Remark. It is better to have a Learner that poses the least Equivalence queries as possible, because it is more expensive to test it than to answer to Membership queries.

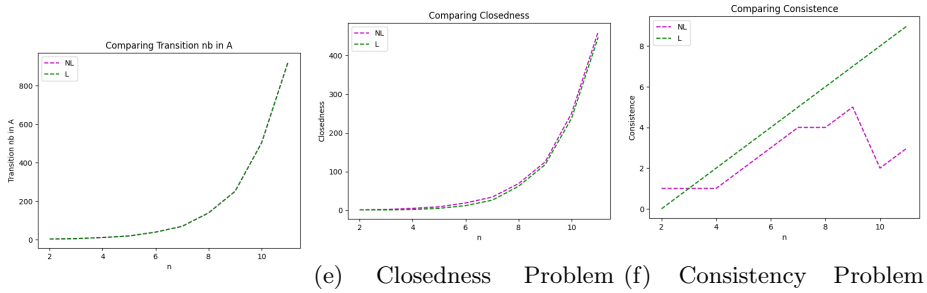
We would also point out that the curve of the Membership queries of NL* in Figure 4b is slightly bigger then the number of L*. In fact, we can see that the number n of states of the *mDFA* equals the number of states of the *cRFSA*, so every state of the *mDFA* is a *Prime* residual. Let n be the size of the *mDFA* (= size of the *cRFSA*). The Learners have to put in S at least n different rows and at least $\log_2(n)$ columns to find the good conjecture. *NL** has to make more membership queries because every time it receives a counter-example from the Teacher, it has to add it together with all its suffixes in E, and since $|S|$ can be big, it may be necessary to make a lot of membership queries to complete every cell of the new-added column. That's why also in Figure 4e the curve of *NL** is slightly worst than the *L** curve.

Finally, it is interesting to see that the number of transition of the *cRFSA* is exactly the same as the number of transitions in the *mDFA*. But in general, and we will try to show it in the next section, the number of transitions in a *cRFSA* is bigger then the number of transitions in the *mDFA*.

[13] Note that in Figure 3c the curve of NL* and L* are superposed



(a) Comparing State Number (b) Membership queries Number (c) Equivalence queries Number



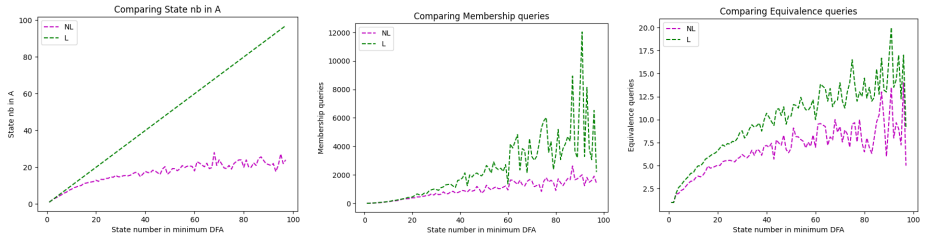
(d) Transition Number (e) Closedness Problem Number (f) Consistency Problem Number

Figure 4: mDFA vs cRFSA on $\mathbf{U} = L(A_n)$

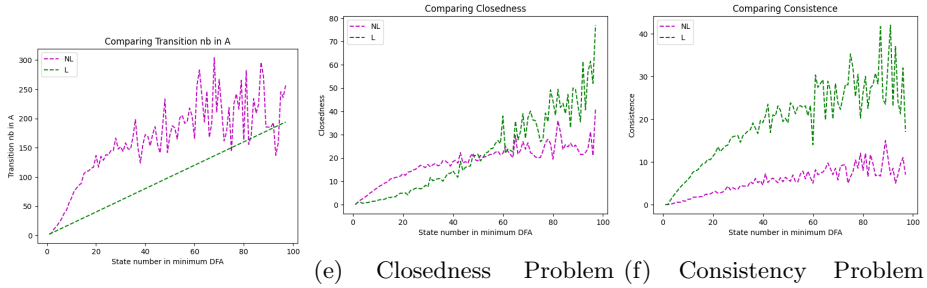
7.3 Results over random Teachers

The third type of comparison has been performed over a huge number of *Teachers* on the form of *mDFA* of size varying from 1 to 100 states. In this way we are able to see in average the performances of L versus NL^*.

This benchmark has been done taking the *Automata* from the *GitHub* repository https://github.com/parof/buchi-automata-benchmark which samples around 2000 examples. Every automaton is supposed to represent a *Büchi Automaton* over a binary alphabet and we have reused them to create *mDFA*. We have transformed then every *Automaton* of the list into a *Teacher* and, one by one, every *Teacher* has been submitted to L^* and NL^* to get at the end some statistics that we could compare to those proposed by Bollig *et al.* in [4].



(a) Comparing State Number (b) Membership queries Number (c) Equivalence queries Number



(d) Transition Number (e) Closedness Problem Number (f) Consistency Problem Number

Figure 5: mDFA vs cRFSA on random Teachers

We see in Figure 5 that the number of states of the *cRFSA* is in general exponentially smaller than the *mDFA* and that, the number of equivalence and membership queries is in general taken by NL^*. As

expected the number of Consistency problems is won by *NL** but curiously the Closedness comparison is taken by *NL** only after about 50 states.

Looking Figure 5b and Figure 5c, we can see that the gap between *NL** and *L** curves is not really big. Again, as for the previous section, when dealing with Teacher which demands a similar number of Membership and Equivalence queries for both *NL** and *L**, it is *NL** which will have more Closedness problems. This is due to the fact that *NL** have to find all the *Prime* residuals and so it will have to promote more rows, thing that is more efficiently done by *L** which adds the counter-example in S.

In Figure 5d we see that the number of transitions of the *mDFA* grows linearly with respect to the number of its states[14], but the number of transitions in the *cRFSA* is often bigger. This is due to the fact that the *cRFSA* tends to create a lot of transitions, sometimes redundant, from a state q_i to every state q'_i where $L(q'_i) \subseteq L(q_i)$.

8 Some curiosity about the two algorithms

While discussing with my supervisors we have seen that the *L** or the *NL** algorithms can have some peculiar properties.

8.1 Minimize an automaton with L*?

If in a certain way, we merge the notion of Teacher and Learner L* together we can create an algorithm *Minimizer* to calculate *mDFA* of a general *DFA*. The *Minimizer* will be composed by the L* implementation and a second part able to compare two automata and provide as the Teacher a counter-example.

We have seen that if we create a *Minimizer* the algorithm will be something like this:

Algorithm 2: Minimizer

Input: An automaton $\mathcal{A}$
Output: $\mathcal{A}$ minimized

```
1  while True do
2  |  Run L*;
3  |  if Membership Query then
4  |  |  Check if ω is accepted by L*;
5  |  else
6  |  |  Check the equivalence between the conjecture A' and the A;
7  |  |  if Equivalence then
8  |  |  |  return A'
9  |  |  else
10 |  |  |  Find a counter-example
11 |  |  end
12 |  end
13 end
```

The time complexity is a bit more complicated to calculate. Let N be the size of the automaton passed in parameter and n the size of its *mDFA*. Then, at most, *L** will create n conjectures, one per state of the *mDFA*. So we will pass through the loop n times. The complexity of the equivalence test between automata is at most:

- $O(n)$ to make to complementation of an automaton of size n;
- $O(n \times m)$ to calculate the intersection of two automata of size n and m;
- $O(n)$ to test if the automaton is empty or find a counter-example.

The equivalence ask for: $(\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{A}')}) \cup (\overline{\mathcal{L}(\mathcal{A})} \cap \mathcal{L}(\mathcal{A}'))$ taking $O(2 \times (N + n + N \times n)) = O(n \times N)$.

The Learner complexity is $O(n^3)$ because the size of the longest counter-example will be at most n[15].

The overall complexity inside the loop is $O(n) * (O(n^3) + O(n \times N)) = O(n^4 + n \times N)$.

The Hopcroft's minimization algorithm minimize an automaton is $O(N \log(N))$ which is clearly better than the *Minimizer* complexity, however the *Minimizer* can be quicker if the automaton to minimize has

[14] Number of transitions = $|\Sigma|\times$ number of states of the Automaton
[15] The counter-example will always be of minimal length

a very huge number of states and its minimized version is very small. We can also create an *NL-Minimizer* but in this case it will work with non-deterministic automata and the operation of complementation on that kind of automata is a hard problem.

8.2 Can we remove the Consistency check?

While testing the performances of the Learners algorithms, I have tried to modify some of their properties to see their behavior. For example I have seen that if we remove the consistency check on the *NL** algorithm, the algorithm seems to terminates on every expression of the benchmark sample. I think that adding the counter-example into E will automatically allow to add the necessary columns to find the *Prime* residuals. Nonetheless I do not have a concrete proof.

In fact the consistency check allows to add new columns, but if we do not add this constraints, we will send conjectures when the table is only closed and the next counter-example will allow to spot new residuals. The main drawback of this approach is that the Learner will have to make more Equivalence queries and the Teacher will have to work harder.

9 Conclusion

In this report, have presented an compared two algorithms on how regular languages can be learned. We can represent them via both minimum deterministic finite state automata or canonical residual finite state automata. The first type of automaton is maybe more understandable for humans: it is easy to check if automaton accepts a word, but on the other side, it can be much more expensive in space. Other studies have been done to seek other way to create automata which can be even smaller then *cRFSAs*, for example, we can have automata that use logical operators on transitions such as the *Alternating Finite State Automata* or other automata using links between symbols of the alphabet. All of these algorithms are an evolution of the L^* algorithm and work on languages which are regular. It should be interesting to see if it is possible to conceive learning algorithms for not regular languages such as context-free grammars. Of course, these implementations would have to output more expressive classes of automata such as the automata working with an auxiliary stack. It is clear, however, that the more complicated the language is, the more powerful will be the returned data-structure representing it. This *TER* has been really compelling. I like automata and the paper I studied allowed me to discover intriguing way to build them from a regular expression. I have also been able to understand the class of *cRFSA* automata that I didn't know before, and, even if it is inside the *NFA* automata world, it is possible to define a unique *canonical* representative for any regular language. Moreover, this *TER* allowed me to see how a researcher works and how satisfying it is to be able to code and test the algorithms studied. I hope I could continue to work in this way in my future, because I think that the theory is a fundamental tool letting you understand how, and above all why, the practical applications of the computer science work.

9.1 Thanks

I would like to thank Ms. Di Giusto and M. Lozes for their support during the development of this study, they have been really helpful to suggest me different approaches to analyze and different point of view to interpret the results I obtained.

References

[1] Dana Angluin. *Learning Regular Sets from Queries and Counterexamples*. Department of Computer Science, Yale University, 1987.

[2] François Denis et al. *Residual Finite State Automata*. LIF, UMR 6166 CNRS, Université de Provence, Marseille, 2002.

[3] John D. Hunter. *Matplotlib*. 2003. URL: https://matplotlib.org/.

[4] Benedikt Bollig et al. *Angluin-Style Learning of NFA*. Ecole Normale Supérieure de Cachan, 2008.

[5] Olivier Carton. *Language Formels, Calculabilité et Complexité*. Ed. by VUIBERT. 2008.

[6] Ivan Zuzak et al. *Noam.js*. 2013. URL: https://github.com/izuzak/noam.

[7] Knut Sveidqvist. *Mermaid.js library*. 2015. URL: https://github.com/mermaid-js/mermaid.

[8] Magnus Jacobsson. *d3-graphviz*. 2018. URL: https://github.com/magjac/d3-graphviz.

A L* algorithm implementation

Algorithm 3: Angluin L* algorithm

Input: Σ
Output: A such that $L(A) = U$

$S \leftarrow \{\varepsilon\}, E \leftarrow \{\varepsilon\}, T \leftarrow$ new Dict();
while $True$ **do**
  if (S, E, T) *is not complete* **then**
    complete(S, E, T) with membership queries;
  end
  if (S, E, T) *is not closed* **then**
    $rowToPromote \leftarrow \{x \in SA$ such that $row(x) \notin upperRows(T)\}$;
    $S \leftarrow S \cup \{rowToPromote\}$;
  else if (S, E, T) *is not consistent* **then**
    Find $s_1, s_2 \in S, x \in \Sigma, e \in E$ such that $row(s_1) = row(s_2)$ and $T(s_1 \cdot x \cdot e) \neq T(s_2 \cdot x \cdot e)$;
    add $x \cdot e$ to E and complete T with membership queries;
  else
    $A \leftarrow createAutomaton(S, E, T)$;
    $teacherAnswer \leftarrow equiv(A)$;
    if *teacherAnswer is a counter example* t **then**
      for $p \in Pref(t)$ **do**
        add p to S and complete this row with membership queries ;
      end
    else
      return A ; `// The Teacher accepted the automaton`
    end
  end
end

B NL* algorithm implementation

Algorithm 4: Bollig *et al.* NL* algorithm

Input: Σ
Output: A such that $L(A) = U$

$S \leftarrow \{\varepsilon\}, E \leftarrow \{\varepsilon\}, T \leftarrow$ new Dict();
while $True$ **do**
  if (S, E, T) *is not complete* **then**
    complete(S, E, T) with membership queries;
  end
  if (S, E, T) *is not RFSA-closed* **then**
    $rowToPromote \leftarrow \{x \in SA$ such that $row(x) \notin upperRows(T)$ and $row(x)$ is Prime$\}$;
    $S \leftarrow S \cup \{rowToPromote\}$;
  else if (S, E, T) *is not RFSA-consistent* **then**
    Find $s_1, s_2 \in S, x \in \Sigma, e \in E$ such that $row(s_1) \sqsubseteq row(s_2)$ and $T(s_1 \cdot x \cdot e) \not\sqsubseteq T(s_2 \cdot x \cdot e)$;
    add $x \cdot e$ to E and complete T with membership queries;
  else
    $A \leftarrow createAutomaton(S, E, T)$;
    $teacherAnswer \leftarrow equiv(A)$;
    if *teacherAnswer is a counter example* t **then**
      for $s \in Suff(t)$ **do**
        add s to E and complete this column with membership queries ;
      end
    else
      return A ; `// The Teacher accepted the automaton`
    end
  end
end

C L* and NL* execution with example

This section aims to provide an example of the execution of the two algorithms taking the unknown language **U** defined by $\varepsilon + (b + aa + abb)(a + b)^*$

C.1 L* Execution

1.

T_1	ε
ε	1
b	1
a	0

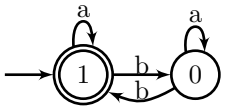
The table is not closed, because $row(a) = 0$ is not present in S.
b will be promoted.

2.

T_2	ε
ε	1
a	0
b	1
ab	0
aa	1

ab and aa have been added in SA to respect the condition where: $\forall x \in S, \forall y \in \Sigma, \exists z \in S_{ext}$ such that $x \cdot y = z$.
The table is closed ($\forall x \in SA, \exists s \in S \mid row(x) = row(s)$) and consistent (there are no two similar rows in S), the *Learner* can send its conjecture.

3.



This is the automaton sent. The states are 0 and 1 since there are two distinct rows in the upper part of the table, 1 is the initial state since $row(\varepsilon) = 0$ and it is also accepting since $T(\varepsilon) = 1$.
Transitions are made as follow:

- $\delta(1, a) = \delta(row(\varepsilon), a) = row(\varepsilon \cdot a) = 0$.
- $\delta(1, b) = \delta(row(\varepsilon), b) = row(\varepsilon \cdot b) = 1$.
- $\delta(0, a) = \delta(row(a), a) = row(a \cdot a) = 1$.
- $\delta(0, b) = \delta(row(a), b) = row(a \cdot b) = 0$.

However, the automaton is not valid, since the word aba is accepted by the conjecture but not by the *Teacher*.

4.

T_3	ε
ε	1
a	0
aba	0
ab	0
b	1
aa	1
abab	0
abaa	0
abb	1

aba as been added in S as well as each of its prefixes.
$abab, abaa$ and abb have been added in SA to keep the *O.T.* complete.
This table is not consistent: $row(a) = row(aba)$ but taking $row(a \cdot a) \neq row(aba \cdot abaa)$. Column a is going to be added because $T(a \cdot a \cdot \varepsilon) \neq row(aba \cdot a \cdot \varepsilon)$

5.

T_4	ε	a
ε	1	0
a	0	1
aba	0	0
ab	0	0
b	1	1
aa	1	1
abab	0	0
abaa	0	0
abb	1	1

$T_4 3$ is not closed: $row(b)$ is not present in S, so it will be promoted.

6.

T_5	ε	a
ε	1	0
a	0	1
aba	0	0
ab	0	0
b	1	1
aa	1	1
abab	0	0
abaa	0	0
abb	1	1
bb	1	1
ba	1	1

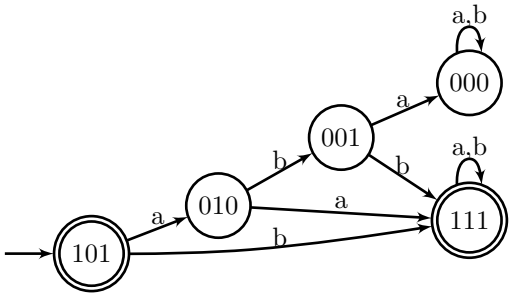
T_5 is not consistent $row(aba) = row(ab)$ but $row(abab) \neq row(abb)$ Column b is going to be added because $T(aba \cdot b \cdot \varepsilon) \neq row(ab \cdot b \cdot \varepsilon)$

7.

T_6	ε	a	b
ε	1	0	1
a	0	1	0
aba	0	0	0
ab	0	0	1
b	1	1	1
aa	1	1	1
abab	0	0	0
abaa	0	0	0
abb	1	1	1
bb	1	1	1
ba	1	1	1

T_6 is closed and consistent.

8.



This automaton recognized perfectly the language proposed by the Teacher, so the algorithm can stop.

C.2 NL* Execution

1.

T_1	ε
*ε[16]	1
*b	1
*a	0

The table is not closed, because $row(a) = 0$ but there is no $s \in S$ such that $row(s) \subseteq 0$. a will be promoted.

2.

T_2	ε
*ε	1
*a	0
*b	1
*ab	0
*aa	1

T_2 is not consistent: $row(a) \sqsubseteq row(\varepsilon)$ but $row(a \cdot a) \not\sqsubseteq row(\varepsilon \cdot a)$. Column a is going to be added because $T(a \cdot a \cdot \varepsilon) \neq row(\varepsilon \cdot a \cdot \varepsilon)$.

[16] Prime rows are indicated by a leading asterisk

3.

T_3	ε	a
*ε	1	0
*a	0	1
*b	1	1
*ab	0	0
*aa	1	1

T_3 is not closed since $row(ab) = 00$ but $\nexists x \in S$ such that $x \sqsubseteq row(ab)$

4.

T_4	ε	a
*ε	1	0
*a	0	1
*ab	0	0
b	1	1
aa	1	1
abb	1	1
*aba	0	0

T_4 is not consistent: $row(ab) \sqsubseteq row(a)$ but $row(ab{\cdot}b) \not\sqsubseteq row(a{\cdot}b)$. Column b is going to be added because $T(ab \cdot b \cdot \varepsilon) \neq row(a \cdot b \cdot \varepsilon)$.

5.

T_5	ε	a	b
*ε	1	0	1
*a	0	1	0
*ab	0	0	1
b	1	1	1
aa	1	1	1
abb	1	1	1
*aba	0	0	0

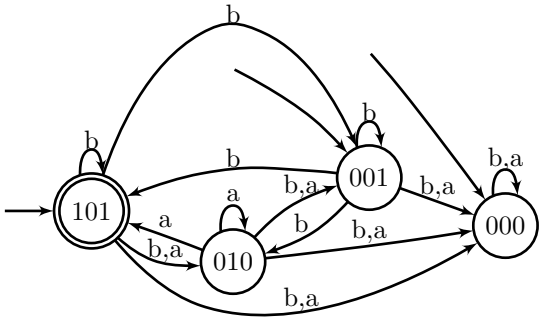
T_5 is not closed since $row(aba) = 000$ but $\nexists x \in S$ such that $x \sqsubseteq row(aba)$.
aba will be promoted.

6.

T_6	ε	a	b
*ε	1	0	1
*a	0	1	0
*ab	0	0	1
*aba	0	0	0
b	1	1	1
aa	1	1	1
abb	1	1	1
*abab	0	0	0
*abaa	0	0	0

T_6 is closed and consistent.
The conjecture will be sent.

7.



Construction of this automaton:

(a) $Q = 101, 010, 001, 000$ which are the Prime rows in S;

(b) $Q_I = 101, 001, 000$ which are all the rows that are covered by $row(\varepsilon)$;

(c) $F = 101$ which is the only prime row having a 1 in the column of ε;

(d) Transitions are more complicated to analyze, we will only describe those starting from the state 101. We have $row(\varepsilon) = 101$ and we have to calculate the transition when reading:

i. $a \to row(\varepsilon \cdot a) = 010$ and the set of rows that are covered by 010 is $010, 000$, so we draw two transitions labelled with a from 101 to 010 and to 000;

ii. $b \to row(\varepsilon \cdot b) = 111$ and the set of rows that are covered by 111 is $010, 000, 101, 001$, so we draw four transitions labelled with a from 101 to 010, 000, 101 and 001.

Moreover, this automaton recognizes precisely the language proposed by the Teacher, so the algorithm can stop.

C.3 Example to find residual from a regular language

In this section we would like to explain why the *mDFA* of Appendix C.1 has one state more of the *cRFSA* of Appendix C.2.

Let's start with the regular expression of $\mathbf{U} = \varepsilon + (b + aa + abb)(a + b)^*$. The residuals of this language can be found iteratively thanks Algorithm 5.

Algorithm 5: Algo to find residuals of a language $\mathbf{U}$

Input: $\mathbf{U}$, Σ
Output: $\mathcal{R}$ = The set of residuals of $\mathbf{U}$

```
1  R ← {} ;                                   // Initialisation of R
2  E ← {ε};                                   // The list of words to treat
3  while E.length > 0 do
4  |  current ← E.pop();
5  |  residual = (current)^-1 L^a;
6  |  if residual ∉ R then
7  |  |  R ← R ∪ {residual};
8  |  |  for sym ∈ Σ do
9  |  |  |  word ← current · sym;
10 |  |  |  E ← E ∪ {word};
11 |  |  end
12 |  end
13 end
14 return R
```

[a] This operation can be performed, for example, with the *Brzozowski derivative* algorithm

Let's apply this algorithm over $\mathbf{U}$.

Iteration	current	Operation	Result	$\mathcal{R}$	E
Init	–	–	–	$\mathcal{R} = \{\}$	$E = \{\varepsilon\}$
1	ε	$\varepsilon^{-1}\mathbf{U}$	$\mathcal{R}_1 = \mathbf{U}$	$\mathcal{R} = \mathcal{R} \cup \mathcal{R}_1$	$E = \{a, b\}$
2	a	$a^{-1}\mathbf{U}$	$\mathcal{R}_2 = (a + bb)(a + b)^*$	$\mathcal{R} = \mathcal{R} \cup \mathcal{R}_2$	$E = \{b, aa, ab\}$
3	b	$b^{-1}\mathbf{U}$	$\mathcal{R}_3 = (a + b)^*$	$\mathcal{R} = \mathcal{R} \cup \mathcal{R}_3$	$E = \{aa, ab, ba, bb\}$
4	aa	$(aa)^{-1}\mathbf{U}$	$(a + b)^* = \mathcal{R}_3$	–	$E = \{ab, ba, bb\}$
5	ab	$(ab)^{-1}\mathbf{U}$	$\mathcal{R}_4 = b(a + b)^*$	$\mathcal{R} = \mathcal{R} \cup \mathcal{R}_4$	$E = \{ba, bb, aba, abb\}$
6	ba	$(ba)^{-1}\mathbf{U}$	$(a + b)^* = \mathcal{R}_3$	–	$E = \{bb, aba, abb\}$
7	bb	$(bb)^{-1}\mathbf{U}$	$(a + b)^* = \mathcal{R}_3$	–	$E = \{aba, abb\}$
8	aba	$(aba)^{-1}\mathbf{U}$	$(a + b)^* = \mathcal{R}_3$	–	$E = \{abb\}$
9	abb	$(abb)^{-1}\mathbf{U}$	$\mathcal{R}_5 = \varnothing$	$\mathcal{R} = \mathcal{R} \cup \mathcal{R}_5$	$E = \varnothing$

We see that the residuals of $\mathbf{U}$ are exactly five, the same number of state of the *mDFA* returned by *L**, but only four of them are *prime* because $\mathcal{R}_3 = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_4$. That's why the *cRFSA* has only four states.