Learning Finite State Automata and Regular Expressions from examples

Vachagan Gratian
Universität Stuttgart
vgratian@utopianlab.am

Abstract

Learnability of languages from examples is a key problem in Computability theory and Computational Linguistics. We describe a pipeline of algorithms to construct a 0-reversible automaton from a finite set of positive examples $\mathcal{S}$ based on the Dana Angluin's Zero-Reversible Inference algorithm (1982) and to extract a regular expression based on the State-Elimination algorithm. While it has been shown that no language can be *identified in the limit* from $\mathcal{S}$ alone, PAC-learning is guaranteed for 0-reversible languages, a subclass of regular languages. We give a detailed overview of the theoretical and practical aspects of this subject. Our implementation, the program `A0LREe`, serves as proof-of-concept and is available online [1].

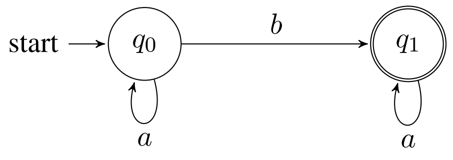
1 Introduction

Our task is to identify a language through a finite set of positive examples. Mathematically, a language $\mathcal{L}$ is just a set of strings composed from some finite alphabet Σ. Then the problem of learnability is, after a finite set of observations $\mathcal{S} = \{s_1\ s_2\ \dots\ s_k \in L\}$, either to induce a valid description of $\mathcal{L}$ or to be able to predict correctly whether a subsequent set of observations $s_{k+1}\ s_{k+2} \dots s_n$ are members of $\mathcal{L}$ or not. When correct prediction is done infinitely, it is said that $\mathcal{L}$ is *identified in the limit* (Gold's theorem).

In this paper, our assumption is that $\mathcal{L}$ is regular and 0-reversible. For this class of languages, a deterministic finite state automaton (FSA) can serve as a "valid description", which in turn can be converted into regular expression (RE) although there is no guarantee that the latter will always have a finite length.

[1] See `https://github.com/vgratian/A0LREe`.

Figure 1



(a) FSA of the language a^*ba^*

As an illustration, consider the set of examples from an unknown language $\mathcal{L}$:

$$\mathcal{S} = \{b, baaa, ab, aba, aab, aaaabaaaa\}$$

A reasonable guess would be that $\mathcal{L}$ has the alphabet $\Sigma = \{a, b\}$ and that $\mathcal{L}_h = \{a^*ba^*\}$. The acceptor of this language is shown in Fig. 1a. Note that although our hypothesis is *Probably Approximately Correct* (PAC), we can never claim with absolute certainty that $\mathcal{L}_h$ identifies $\mathcal{L}$, as we shall see later. Moreover, there are an infinite number of FSAs and REs that can be a valid description of the same language. E.g. the expressions

- $a?a? \dots a?ba?a? \dots a?$
- $b|ab|ba|aba|aaba \dots$

both have infinite length, but are isomorphic to a^*ba^*. Obviously, we want to infer a description of $\mathcal{L}$ from $\mathcal{S}$ which is not only PAC, but is also the smallest possible description. As we shall see later, although we are guaranteed to learn an optimally small FSA from $\mathcal{S}$, converting the former to an optimally small RE is notably challenging.

The rest of this paper is organized as follows: in 2, we give a overview of important works related to our subject matter. In 3 we give a shot description of the definitions and mathematical concepts used in the paper. In 4 we describe the algorithms of the AOLREe pipeline. We conclude with a brief discussion in **??**.

2 Related Work

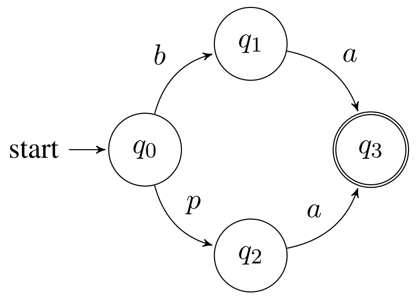
Learnability of languages has been extensively researched over the past half century. Both, its theoretical and practical aspects are a pivotal issue in Computer Science. Those include inference, identification and evaluation of learning algorithms. In case of inference, the computational feasibility of learning algorithms is of high importance. Generally, an algorithm is considered feasible if it runs in polynomial time. Secondly, we want some proof that the inferred hypothesis (an FSA, RE, Turing machine, tree grammar, SCFG, etc.) correctly or "reasonably" identifies the unknown $\mathcal{L}$. Finally, we need some criteria to asses weither our hypothesis is an optimal representation of $\mathcal{L}$: preferably it is not only a correct, but also the smallest or simplest possible representation.

Perhaps the most important theoretical work is Gold's 1967 paper [1] whose significance is in particular the concept of *identification in the limit* that has been widely adopted in the field. Gold defines learning as an infinite process where the learner is provided strings $s_t, s_{t+1}, s_{t+2} \dots s_\infty$ at each timepoint t, has to guess whether or not it is a member of $\mathcal{L}$ and gets the correct answer directly afterwards. Thus, in this scenario, $\mathcal{S}$, can contain both positive and negative examples. At each t, the learner constructs a model for the language, M_t, that is, his or her hypothesis at timepoint t. Gold states, that the learner has identified $\mathcal{L}$ in the limit if:

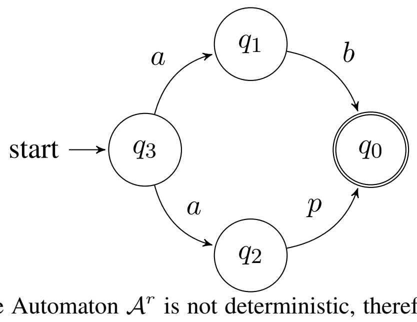
- after some finite time k, M is not changed anymore (i.e. $M_k = M_{k+1} = M_{k+2} \dots$), and
- the learner makes infinitely many correct guesses afterwards.

Gold points out that the second condition is necessary for the following reason. If the learner makes only finitely many correct predictions, it is always possible that the next observed example will contradict M_k and therefore they do not guarantee correct identification of $\mathcal{L}$. Moreover, at any $k = t < \infty$, there will be infinitely many languages that could have generated $s_1 \dots s_k$, but M_k can only be one of them. Gold also discusses an *active* learning scenario where the learner is not

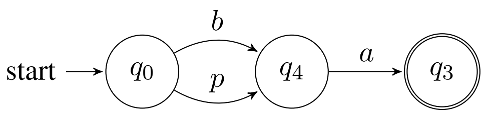
Figure 2: The language $\mathcal{L} = \{(b|p)a\}$ is zero-reversible



(a) The Automaton $\mathcal{A}$ accepts $\mathcal{L}$ and is deterministic.



(b) The Automaton $\mathcal{A}^r$ is not deterministic, therefore $\mathcal{A}$ is not zero-reversible.



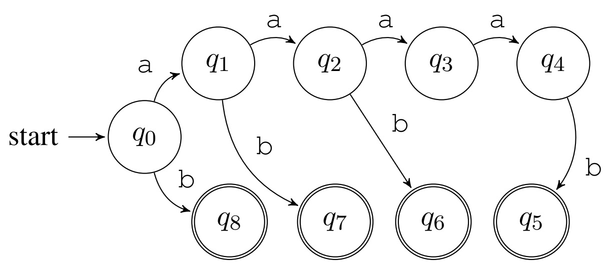
(c) We can however convert $\mathcal{A}$ into the zero-reversible automaton $\mathcal{A}_0$ by merging states q_1 and q_2 into a new state q_4.

only provided with examples from $\mathcal{L}$, but additionally makes membership queries to an *informant* (also named *oracle* in literature). Gold demonstrates that the class of context-sensitive languages is learnable in this scenario. Conversely, only a limited part of regular languages is learnable from examples only (i.e. *passive* learning).

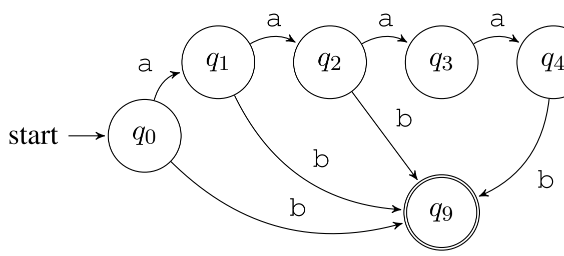
The notion of active learning has been extended by other authors with other types of queries, such as the *equivalence query* (learner can ask if the hypothesis $M = \mathcal{L}$), the *correction queries* (if hypothesis is incorrect oracle provides a correcting example) [2] [3].

In another paper [4], Gold demonstrates that finding the smallest possible FSA which agrees with some finite $\mathcal{S}$ is NP-complete. Learnability in more general terms is discussed in [5] where the domain of discourse are sets of propositional concepts and the author demonstrates that only a small class of these sets are "feasibly learnable" from positive examples, i.e. an algorithm could

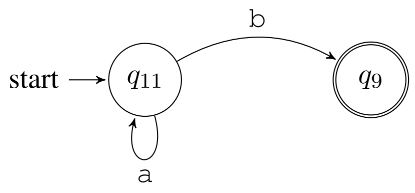
Figure 3: Stages of constructing $\mathcal{A}_0$ from $\mathcal{S} = \{ab, aab, aaaab, b\}$



(a) Prefix Tree



(b) Merged final states of $\mathcal{A}_P$ ($q_5, q_6, q_7, q_8 \rightarrow q_9$)



(c) Merging non-deterministic edges results in $\mathcal{A}_0$ ($q_0, q_1, q_2, q_4 \rightarrow q_{11}$, i.e. all b-predecessors of q_9)

"closely approximate" them in polynomial time. For more background on automata, learnability and languages, we refer to [6] [7] [8]. Of high importance to the subject are the works of Dana Angluin ([9] [10] [11] [12] [13]) who is one the earliest authors to address the learnability of languages and automata and introduced a number of important theorems and algorithms. In particular, in [11], Angluin introduces two algorithms for learning FSAs from positive examples in near-linear and polynomial time respectively. It includes also comprehensive theoretical body of work, including proof that for each 0-reversible language $\mathcal{L}_0$ there exists a finite *characteristic sample* $\mathcal{S}$ such that it is the smallest subset of $\mathcal{L}_0$ in which $\mathcal{L}_0$ can be identified in the limit [2]. The two algorithms introduced in the paper are the Zero-Reversible Inference Algorithm and the k-Reversible Inference Algorithm. Our work is an implementation of the former.

Algorithms for learning stochastic languages through Weighted Finite-State Automata (WFAs) have been introduced by other authors, see for example [14] for an overview. Those however lie outside the scope of this paper. Learning algorithms of Automata (both FSAs and WFAs) have an important application in many machine learning tasks, such as computer vision, natural language processing and bioinformatics [15] [16] [17] [18]. They serve as basis for statistical machine learning models, such as Hidden Markov Models [19]. Theories about automata and learnability also have important implications for cognitive science, especially with regard to the problem of language acquisition (see e.g. [20] [21] [22]).

While Automata can be seen as a graphical representations of a (regular) languages, regular expressions (REs) on the other hand, can be seen as a "flat" representation of the whole language. REs are sequences of alphabetic symbols with disjunctions, conjunctions and closures. The concept was first introduced by Kleene in the 1950s [23], and is since then widely adopted in many computer applications. For example, the Linux program `grep` and similar tools convert REs into FSAs or NFAs (non-deterministic FSAs) to do string matching in text [24].

Converting REs into FSAs is computationally trivial, however no optimal solutions exists for the opposite task. Since a FSA expresses a language in a "compact" way, it is possible that an n-state automaton, where n is very small, is only expressible with a very large or even infinite regular expression. Theoretical and practical aspects of this task are discussed in [25] [26] [27]. One algorithm for extracting REs, is the State Elimination model, that has been adopted in our work. We have followed the description in [26].

3 Preliminaries

3.1 Automata

The automaton $\mathcal{A}$ is an adequate description of some language $\mathcal{L}$. Specifically, it defines the set of symbols (the alphabet) of $\mathcal{L}$ and the set of formation rules of the strings of $\mathcal{L}$. We denote the language that corresponds to the automaton as $\mathcal{L}(\mathcal{A})$ and the automaton that accepts $\mathcal{L}$ as $\mathcal{A}_{\mathcal{L}}$. Given a string s, the automaton will accept it if $s \in \mathcal{L}$ and reject otherwise.

The graphical depiction of an automaton, as is done in Fig. 1a, is only for the ease of understand-

[2] Note that, following Gold's theorem, even if $\mathcal{L}_0$ can be identified in the limit, we would not be able to prove that that is the case since $\mathcal{S}$ is finite.

ing for humans, and is not a mathematical property. Note that if we think of the alphabet as the set of axioms of $\mathcal{L}$ and the formation rules as its rules of inference, it becomes clear that an automaton is a valid formal system. Hence, all strings in $\mathcal{L}$ are theorems of $\mathcal{L}$ if and only if they are accepted by $\mathcal{A}_{\mathcal{L}}$.

More formally, $\mathcal{A}$ is the tuple $(\Sigma, Q, I, F, \boldsymbol{\delta})$, such that:

- Σ is the set of symbols, optionally including the empty string ϵ,
- Q is the set of states: $q_0, q_1 \dots q_n$,
- $I \subseteq Q$ is the set of initial states,
- $F \subseteq Q$ is the set of final or accepting states,
- $\boldsymbol{\delta}$ is the $Q \times Q \times \Sigma$ transition matrix, that maps $\delta(q_i, x) \rightarrow \{q_j\} \cup \varnothing$ for $\forall q_i, q_j \in Q$ and $\forall x \in \Sigma$. [3]

Note that when we depict $\mathcal{A}$ as a graph, the states will be the *nodes* and the transition matrix the *edges* of the graph. For example, the graph depicted in Fig. 1a, is the automaton:

- $\Sigma = \{a, b\}$
- $Q = \{q_0, q_1\}$
- $I = \{q_0\}$
- $F = \{q_1\}$
- $\boldsymbol{\delta} = \begin{bmatrix} a & b \\ \varnothing & a \end{bmatrix}$

3.2 Deterministic Automata

An automaton with n states, where $n \in \mathbb{N}$, is a *finite-state* automaton (FSA). An automaton is *deterministic* (DFA) if and only if it has no state with more than one outgoing edge for the same symbol $x \in \Sigma$. It is possible to convert a *non-deterministic* (NFA) automaton into a DFA, as we will see in 4.

The automaton in Fig. 1a is both finite and deterministic.

3.3 Languages

Mathematically a language is simply a set *strings*. A string itself is a set of symbols from some alphabet Σ [4]. $\mathcal{L}$ might be finite or infinite. In fact, even over a finite alphabet Σ, infinitely many languages exist.

A language is *known* or identified if we either know all elements of the set or we posses an ade-

Algorithm 1

```
function PREFIXTREECONSTRUCTOR
    input: S = list of strings
    output: Prefix Tree

    $S \leftarrow \texttt{sort}(S)$
    $N, E, F, I, p \leftarrow \varnothing$
    $root \leftarrow \texttt{Node}()$
    $N \leftarrow N \cup root$
    $I \leftarrow I \cup root$
    if $\epsilon \in S$ then
        $F \leftarrow F \cup root$
        $S \leftarrow S \setminus \epsilon$
    for all $s \in S$ do
        $p \leftarrow \texttt{MatchPrefixPath}(p, s)$
        $i \leftarrow \texttt{len}(p)$
        while $i < \texttt{len}(s)$ do
            $pnode, plabel \leftarrow p_{-1}$
            $node \leftarrow \texttt{Node}()$
            $N \leftarrow N \cup node$
            if $i + 1 = \texttt{len}(s)$ then
                $F \leftarrow F \cup node$
            $E \leftarrow E \cup \langle pnode, s_i, node \rangle$
            $p \leftarrow p \cup \langle node, s_i \rangle$
            $i \leftarrow i + 1$
    return $\langle N, E, F, I \rangle$

function MATCHPREFIXPATH
    input: $p$ = list of $\langle node, char \rangle$ pairs
           $s$ = string
    output: $new\_p$

    $new\_p \leftarrow \varnothing$
    $j \leftarrow 0$
    while $j < \texttt{len}(p)$ and $j < \texttt{len}(s)$ do
        $node, char \leftarrow p_j$
        $j \leftarrow j + 1$
        if $s_j \neq char$ then
            break
        else
            $new\_p \leftarrow new\_p \cup \langle node, char \rangle$
    if $new_p = \varnothing$ then
        return $new\_p \cup \langle root, \epsilon \rangle$
    else
        return $new\_p$
```

[3]The empty set $\varnothing$ means there is no transition between the two states.

[4]Of course, Set Theory does not permit sets of alphabetical symbols. But since those symbols do not contain meaning in themselves, we can just pair them up with the set of natural numbers. I.e. $\mathcal{L} \in \mathbb{N}^{\mathbb{N}^k}$ where $k = |\Sigma|$.

quate description of it, such as an FSA or RE (if $\mathcal{L}$ is regular). A language is *decidable* or recursive if there is an *effective way* to decide if $s \in \mathcal{L}$ for some $s \in \Sigma^*$. *Effective* means here that a finite number of calculations are required for this decision. Even if we know $\mathcal{A}$, the language might not be decidable [5]. And even if the language is decidable, it might not be *feasibly computable* (e.g. because of exponential growth of computational complexity). Note that decidability is a pure deductive process. *Learnability*, on the other hand, is an inductive process. In this case the language is unknown, but we know $\mathcal{S} \subseteq \mathcal{L}$ and we want to identify $\mathcal{L}$ from the former, e.g. by constructing a corresponding $\mathcal{A}$.

3.4 Prefix and Tail sets

We finally define two functions over $\mathcal{L}$ as introduced by Angluin.

- The set of prefixes of $\mathcal{L}$:
  $\mathcal{P}(\mathcal{L}) = \{\mathtt{x} : \exists \mathtt{y}(\mathtt{xy}, \mathtt{y} \in \mathcal{L})\ \}$
- The set of "tails" of $\mathcal{L}$ and the x:
  $\mathcal{T}_{\mathcal{L}}(\mathtt{x}) = \{\mathtt{x} : \exists \mathtt{z}(\mathtt{xz} \in \mathcal{L})\}$

For illustration, let $\mathcal{S}$ as in Section 1., then:

- $\mathcal{P}(S) = \{a, aa, aaa\}$
- $\mathcal{T}_S(b) = \{a, aa\}$
- $\mathcal{T}_S(a) = \{b, ba, aab, abaa\}$
- $\mathcal{T}_S(aa) = \{ab, baa\}$
- etc.

3.5 Reversible Languages

The notion k-reversibility refers to the morphosyntactic complexity of regular sets. We will only consider here the class of 0-reversible languages. Informally, strings of 0-reversible languages are composed only of prefixes and suffixes and no infixes. In other words, strings can have the form $u_i v_j$, but not $u_i v_j w_k$. Or, as Angluin defines it, a language is 0-reversible iff $\mathcal{T}(u_1 v) = \mathcal{T}(u_2 v)$ for strings $u_1 v, u_1 \in \mathcal{L}$.

Furthermore, Angluin defines that for any 0-reversible language, then there exists a 0-reversible automaton accepting it. An automaton, in turn, is 0-reversible iff:

- it has at most one final state,
- is deterministic,
- is *reset-free*, which means its reversed automaton, $\mathcal{A}^r$ is also deterministic.

As a simple example, the language $\mathcal{L} = \{ba, pa\}$, which contains only two strings, is 0-reversible, because a 0-reversible automaton accepting it does exists as demonstrated in Fig. 2.

[5] Simple example: let $\pi \in \mathcal{L}$ or let $\mathcal{L} = \mathbb{R}$.

Algorithm 2

```
function MakeZeroReversible
    input: Prefix tree = $\langle \Sigma, N, E, F, I \rangle$
    output: Zero-reversible Automaton

    if len($F$) $> 1$ then
        MergeStates($F$)

    for all $node \in N$ do
        for all $char \in \Sigma$ do
            $P, C \leftarrow \varnothing$
            for all $edge \in E$ do
                $source, label, target \leftarrow edge$
                if $label = char$ then
                    if $target = node$ then
                        $P \leftarrow P \cup source$
                    if $source = node$ then
                        $C \leftarrow C \cup target$
            if len($P$) $> 1$ then
                MergeStates($P$)
            if len($C$) $> 1$ then
                MergeStates($C$)

    if NotZeroReversible() then
        MakeZeroReversible($\Sigma, N, E, F, I$)
```

4 Algorithms

In this sections we describe the pipeline of algorithms implemented in the program A0LREe (**A₀**-**L**earner and **R**egular **E**xpression **e**xtractor). The input of the program is a file with a list of strings. The output is a regular expression. Optionally graphs are drawn for illustrating the workflow using the open-source package graphviz.

In the first stage, a Prefix Tree acceptor $\mathcal{A}_P$ is constructed from the input $\mathcal{S}$. In the second stage, the program looks for states that are identical predecessors or successors of some other state and merges them to get a 0-reversible acceptor $\mathcal{A}_0$. Those two stages are based on Angluin's Zero-Reversible Inference algorithm. Finally, the State-Elimination algorithm is applied to derive one single regular expression from $\mathcal{A}_0$. In this stage, all except the intial and final states of the acceptor are eliminated iteratively, such that only one edge remains at the end and the label of this edge is the final regular expression.

4.1 Prefix Tree Constructor

Algorithm 1 constructs an automaton that accepts exactly $\mathcal{S}$. Our implementation includes a heuristic method to avoid creating repeating prefix paths (i.e. sequences of states with same edges). This is done by and keeping track of the previous path in the variable `p` (a list of node-character tuples).

Before starting $\mathcal{S}$ is alphabetically sorted. Then the program starts constructing the automaton by iterating over each string s in $\mathcal{S}$. The function *MatchPrefixPath* will update `p` to keep only the node-character pairs that are the longest match with the characters in s from the left-hand side. Subsequently the algorithm iterates over each remaining character in s, creates a new node and adds a new edge to it from the last node in p (p_{-1}).

The prefix automaton is represented by the four lists of nodes, edges, final states and initial states respectively (`N, E, F, I` in the pseudocode).

4.2 Converting $\mathcal{A}_P$ to 0-Automaton

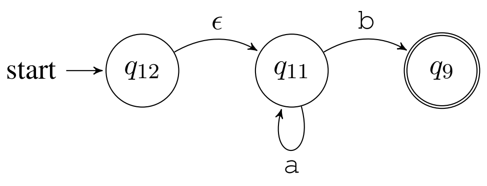
Algorithm 2 converts the prefix tree into a 0-reversible automaton. This is done in two stages. First, all final states are merged into one. Secondly any nodes, $n_1 \dots n_k$ that are a-predecessors or a-successors (for any $a \in \Sigma$) of some other node n_j are merged into one.

This is done by iterating over all nodes, all characters in Σ, and all edges. If a node has more than one predecessor or successor for the same character, those predecessors or successors are merged into one node respectively. Note that the pseudocode has three nested loops, however this algorithm is more efficiently implemented in `A0LREe`, where transitions of the automaton are kept in a matrix and not as a list of tuples as implied in the pseudocode.

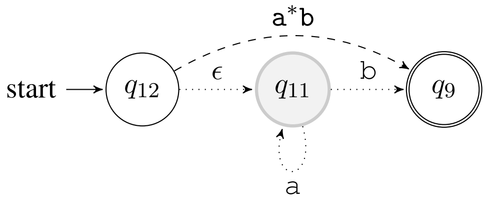
The functions is repeated until the automaton is 0-reversible.

An example of 0-Automaton construction for the input $\mathcal{S} = \{ab, aab, aaaab, b\}$ is shown in Fig. 3. Fig. 3a is the Prefix Tree generated by Algorithm 1. Fig 3b shows the Automaton after merging the final states. This automaton is still not 0-reversible since the states q_0, q_1, q_2, q_4 are all b-predecessors of state q_9. After merging these

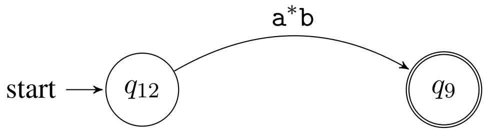
Figure 4: Extracting RE from $\mathcal{A}_0$ of Figure 4



(a) Converted to uniform automaton from $\mathcal{A}_0$ of Figure 4c



(b) Eliminating q_{11} and adding the new edge $q_{12} \to q_9$



(c) State-Elimination halts when only two (initial and accepting) are left. RE = `a`*`b`

states into a new state, q_{11}, we get the 0-reversible Automaton in Fig. 3c.

4.3 Extracting RE from $\mathcal{A}_0$

Algorithm 3 and 4 are the State-Elimination Algorithm as described in [26]. The input of the Algorithm is *unform* Automaton and the output is a regular expression.

4.3.1 Uniform Automaton

The former is defined as an Automaton which:

- is deterministic
- has exactly one initial state with no incoming edges
- has exactly one final states with no outgoing edges

Note that the $\mathcal{A}_0$ produced by Algorithm 2 always satisfies the first, but not always the second and third conditions (e.g., see Fig. 3c, where the initial state has a selfloop edge). Therefore before starting the state-elimination procedure, $\mathcal{A}_0$ is converted into $\mathcal{A}_{0U}$. This is a trivial task, since we only need to add a new initial and/or a final node, and add two ϵ-edges to the old initial and from the old final states respectively (which then become ordinary states). As an example, the automaton of Fig. 3c is converted into a uniform automaton in Fig. 4a.

4.3.2 State Elimination

The function `ReGexParser` (Algorithm 3) iteratively selects a arbitrary state k and eliminates it until only two states are left in the Automaton: the initial and final states. Note that after each iteration a new Automaton is created since E, the set of edges, is completely reconstructed.

Deriving new edges is done by the function `DeriveEdge` (Algorithm 4). If n_k is the state being eliminated, and n_i, n_j are any states of $\mathcal{A}_{0U}$, then the new edge $n_i \to n_j$ is the union of the previous *direct* path from n_i to n_j and the *indirect* path from n_i to n_j through n_k. More formally:

$$\hat{E}(n_i, n_j) = E(n_i, n_j) \vee [E(n_i, n_k) \wedge E(n_k, n_k)^* \wedge E(n_k, n_j)]$$

Note that if neither a direct or indirect path existed from $n_i \to n_j$, the function returns $\varnothing$ and $\hat{E}(n_i, n_j)$ is not added to the new automaton.

As an illustration see Fig. 4. The Automaton contains only three states, so only one state, q_{11}, needs to be eliminated. There is no direct path between nodes q_{12} and q_9. so $E(q_{12}, q_9) = \varnothing$. However there is an indirect path through q_{11} and the algorithm will construct a new edge by taking the intersection of the edges in the indirect path:

$$\begin{aligned}\hat{E}(q_{12}, q_9) &= E(q_{12}, q_9) \vee [E(q_{12}, q_{11}) \\ &\quad \wedge E(q_{11}, q_{11})^* \wedge E(q_{11}, q_9)] \\ &= \varnothing \vee [\epsilon \wedge a^* \wedge b] \\ &= [a^* \wedge b]\end{aligned}$$

Since no more states are left for elimination, the output of the program will be a^*b.

5 Conclusion and Future Work

As we alluded in 1, a major weakness of the `AOLREe` pipeline is that it might produce quite lengthy and complex expressions for even the simplest languages. For example, the program output for

$$\mathcal{S} = \{b,\ abc,\ aabc,\ abcc\}$$

is the expression $a^*(b|bc^*c)$ which is isomorphic to a^*bc^*, but is redundantly long. In our future work we would like to investigate to possible approaches to overcome this weakness. Either a new, joint algorithm that will produce the shortest possible RE from $\mathcal{S}$. Or an algorithm to efficiently merge REs during the State-Elimination process.

Algorithm 3

```
1: function ReGexParser
2:     input: $A = \langle \Sigma, N, E, F, I \rangle$
3:     output: $expression$ = string
4:
5:     $A \leftarrow$ MakeUniform($A$)
6:     $\hat{N} \leftarrow N \setminus F, I$
7:
8:     while len($\hat{N}$) $\neq 0$ do
9:         $n_k =$ pop($\hat{N}$)
10:        $\hat{E} = \varnothing$
11:        for all $n_i \in N \setminus n_k$ do
12:            for all $n_j \in N \setminus n_k$ do
13:                $e \leftarrow$ DeriveEdge($n_i, n_j, n_k$)
14:                if $e \neq \varnothing$ then
15:                    $\hat{E} \leftarrow \hat{E} \cup \langle n_i, e, n_j \rangle$
16:        $N \leftarrow N \setminus n_k$
17:        $\hat{N} \leftarrow \hat{N} \setminus n_k$
18:        $E \leftarrow \hat{E}$
19:
20:    $\langle n_i, e, n_f \rangle \leftarrow$ pop($E$)
21:    return e
```

Algorithm 4

```
1: function DeriveEdge
2:     input: $source, target, k$ = nodes
3:     output: $expression$ or $\varnothing$
4:
5:     $s2t =$ GetEdgeLabel($source, target$)
6:     $s2k =$ GetEdgeLabel($source, k$)
7:     $k2k =$ GetEdgeLabel($k, k$)
8:     $k2t =$ GetEdgeLabel($k, target$)
9:
10:    if len($s2k$) $\neq 0$ and len($k2t$) $\neq 0$ then
11:        if len($s2t$) $\neq 0$ then
12:            return $s2t \vee (s2k \wedge (k2k)^* \wedge k2t)$
13:        else
14:            return $s2k \wedge (k2k)^* \wedge k2t$
15:    else
16:        if len($s2t$) $\neq 0$ then
17:            return $s2k$
18:        else
19:            return $\varnothing$
```

References

[1] E. M. Gold, “Language identification in the limit,” *Information and Control*, vol. 10, no. 5, pp. 447 – 474, 1967.

[2] C. Tîrnăucă and T. Knuutila, “Polynomial time algorithms for learning k-reversible languages and pattern languages with correction queries,” in *Algorithmic Learning Theory* (M. Hutter, R. A. Servedio, and E. Takimoto, eds.), (Berlin, Heidelberg), pp. 272–284, Springer Berlin Heidelberg, 2007.

[3] L. Becerra-Bonache, A. H. Dediu, and C. Tîrnăucă, “Learning dfa from correction and equivalence queries,” in *International Colloquium on Grammatical Inference*, pp. 281–292, Springer, 2006.

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

[5] L. G. Valiant, “A theory of the learnable,” in *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC ’84, (New York, NY, USA), pp. 436–445, ACM, 1984.

[6] M. Kearns, U. Vazirani, V. V, U. Vazirani, and M. Press, *An Introduction to Computational Learning Theory*. The MIT Press, MIT Press, 1994.

[7] J. Hartmanis and R. Stearns, *Algebraic structure theory of sequential machines*. Prentice-Hall international series in applied mathematics, Prentice-Hall, 1966.

[8] J. Hopcroft, *Introduction to Automata Theory, Languages, and Computation*. Always Learning, Pearson Education, 2008.

[9] D. Angluin, “Inductive inference of formal languages from positive data,” *Information and control*, vol. 45, no. 2, pp. 117–135, 1980.

[10] D. Angluin, “A note on the number of queries needed to identify regular languages,” *Information and control*, vol. 51, no. 1, pp. 76–87, 1981.

[11] D. Angluin, “Inference of reversible languages,” *Journal of the ACM (JACM)*, vol. 29, no. 3, pp. 741–765, 1982.

[12] D. Angluin and C. H. Smith, “Inductive inference: Theory and methods,” *ACM Computing Surveys (CSUR)*, vol. 15, no. 3, pp. 237–269, 1983.

[13] D. Angluin, “Learning regular sets from queries and counterexamples,” *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[14] B. Balle and M. Mohri, “Learning weighted automata,” in *International Conference on Algebraic Informatics*, pp. 1–21, Springer, 2015.

[15] M. Mohri, F. Pereira, and M. Riley, “Weighted automata in text and speech processing,” *arXiv preprint cs/0503077*, 2005.

[16] M. Mindek, “Finite state automata and image recognition.,” in *Dateso*, pp. 141–151, 2004.

[17] K. Knight and J. May, *Applications of Weighted Automata in Natural Language Processing*, pp. 571–596. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[18] C. Allauzen, M. Mohri, and A. Talwalkar, “Sequence kernels for predicting protein essentiality,” in *Proceedings of the 25th international conference on Machine learning*, pp. 9–16, ACM, 2008.

[19] P. Dupont, F. Denis, and Y. Esposito, “Links between probabilistic automata and hidden markov models: probability distributions, learning models and induction algorithms,” *Pattern recognition*, vol. 38, no. 9, pp. 1349–1371, 2005.

[20] K. Johnson, “Gold’s theorem and cognitive science,” *Philosophy of Science*, vol. 71, no. 4, pp. 571–592, 2004.

[21] R. C. Berwick and S. Pilato, “Learning syntax by automata induction,” *Machine Learning*, vol. 2, pp. 9–38, Mar 1987.

[22] S. F. Pilato and R. C. Berwick, “Reversible automata and induction of the english auxiliary system,” in *Proceedings of the 23rd Annual Meeting on Association for Computational Linguistics*, ACL ’85, (USA), p. 70–75, Association for Computational Linguistics, 1985.

[23] S. C. Kleene, “Representation of events in nerve nets and finite automata,” tech. rep., RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[24] T. Abou-Assaleh and W. Ai, “Survey of global regular expression print (grep) tools,”

Citeseer. Topics in Program Comprehension (CSCI 6306). Halifax, Nova Scotia, Canada, 2004.

[25] H. Gruber and M. Holzer, “From finite automata to regular expressions and back—a summary on descriptional complexity,” *International Journal of Foundations of Computer Science*, vol. 26, no. 08, pp. 1009–1040, 2015.

[26] S. Getir, D. A. Vu, F. Peverali, D. Strüber, and T. Kehrer, “State elimination as model transformation problem.,” in *TTC@ STAF*, pp. 65–73, 2017.

[27] H. Gruber and M. Holzer, “Provably shorter regular expressions from deterministic finite automata,” in *Developments in Language Theory* (M. Ito and M. Toyama, eds.), (Berlin, Heidelberg), pp. 383–395, Springer Berlin Heidelberg, 2008.