Proceedings of Machine Learning Research 217:59–79, 2023 ICGI regular paper

Learning Syntactic Monoids from Samples by extending known Algorithms for learning State Machines

Simon Dieck S.DIECK@TUDELFT.NL
Sicco Verwer S.S.VERWER@TUDELFT.NL
Technische Universiteit Delft

Editors: François Coste, Faissal Ouardi and Guillaume Rabusseau

Abstract

For the inference of regular languages, most current methods learn a version of deterministic finite automata. Syntactic monoids are an alternative representation of regular languages, which have some advantages over automata. For example, traces can be parsed starting from any index and the star-freeness of the language they represent can be checked in polynomial time. But, to date, there existed no passive learning algorithm for syntactic monoids. In this paper, we prove that known state-merging algorithms for learning deterministic finite automata can be instrumented to learn syntactic monoids instead, by using as the input a special structure proposed in this paper: the interfix-graph. Further, we introduce a method to encode frequencies on the interfix-graph, such that models can also be learned from only positive traces. We implemented this structure and performed experiments with both traditional data and data containing only positive traces. As such this work answers basic theoretical and experimental questions regarding a novel passive learning algorithm for syntactic monoids.

1. Introduction

Many problems can be interpreted as regular languages and accordingly inferring the grammar of such regular languages from samples is a field of significant interest (Matoušek et al., 2021; Aichernig et al., 2018; Ali et al., 2021). Most methods learn these grammars in the form of deterministic finite automata (**DFA**) or their probabilistic versions (**PDFA**). However, there are some limitations in representing the grammars as **DFA**. For example, the prefix-oriented nature of **DFA** leads learning algorithms to also better learn grammars for patterns appearing at earlier indices than those appearing later in traces. For another area of interest in grammatical inference, star-free languages (Jäger and Rogers, 2012), checking for a **DFA**, if the language it represents is star-free, is P-Space complete (Stern, 1985). In this paper we propose learning an alternative representation of regular languages: syntactic monoids with a finite number of congruence classes (**SMF**). It has already been shown, that **DFA** and **SMF** are equivalent when it comes to what kind of languages they can represent (McNaughton and Papert, 1968). Nevertheless, there are some structural advantages to **SMF**. They allow for extending traces both with prefixes as well as suffixes. Therefore, they group interfixes with the same behaviour as opposed to prefixes like a **DFA** does. Further, Green's relations can be checked in polynomial time, since the $\mathcal{L}$ and $\mathcal{R}$ classes correspond to strongly connected components in the left and right Cayley graph (Clifford and Preston, 1961). This also allows us to check if the learned language is star-free in polynomial time (Schützenberger, 1965; Kilp et al., 2000). One of the big hurdles in using **SMF**

so far has been the lack of efficient methods to learn them. In this paper, we show that they can be learned using the same algorithms employed for learning **DFA**s. This will allow for transferring much of the research, that was invested into improving those algorithms to this new model.

In this work, we will first give a small overview of related works and provide some definitions which are heavily used in this paper. Afterward, we introduce the interfix-graph (**IG**), a structure into which samples are parsed and which serves as the new input to known **DFA** learning algorithms. We also prove, that given such an **IG**, a state merging algorithm will return a structure that can easily be turned into an **SMF**. Finally, we experimentally compare **SMF** and **DFA** that have been learned using the same state-merging algorithm.

Our contributions are as follows:

- Introducing the interfix-graph: a structure into which a dataset of traces can be parsed.
- Proving that a standard state-merging algorithm initialised with an interfix-graph will converge to the right and left Cayley graph of a syntactic monoid.
- Introducing a method to encode frequencies on such a structure, so trace probabilities can be computed, which enables learning from only positive traces.
- Performing experiments, analysing the performance of syntactic monoids learned in such a way on 25 instances of the *stamina* dataset and 16 instances of the *Pautomac* dataset.

1.1. Related Work

There has so far been little published research into learning syntactic monoids. There are the works from Clark (Clark and Eyraud, 2007; Clark, 2010a,b, 2015), who uses syntactic monoids as a basis for learning several language types. However, Clark et al. mostly focus on non-finite syntactic monoids in their work and use the framework of active learning. And the work from Stephan and Ventsov (2001), which focuses on the learnability of certain types of monoids.

As for approaches to learning star-free languages, they normally do not try to determine, whether a learned language is star-free or not, but rather either try to learn a grammar that is guaranteed to have certain characteristics, like being star-free, (Wieczorek and Unold, 2016; Fernau, 2005), or learn a grammar from a language they know is star-free (Bhattamishra et al., 2020; Wang, 2021).

2. Definitions and Notation

In order to use the same algorithms for learning **SMF** and **DFA** it is necessary to be able to represent these structures in the same way. We can do this easily in two ways: View both **FSM** and **DFA** as edge-coloured graphs or view the **FSM** as a **DFA** with a larger alphabet. In this paper, we will use the second viewpoint. This section will introduce the related definitions and notations.

Definition 1 (Free Monoid)
Given a set (alphabet) Σ, we can introduce a monoid on this set with string concatenation as its operation (denoted by $\cdot$). Its elements are all possible finite sequences of elements from Σ. We also include the empty sequence λ, which acts as the identity in this monoid. We denote this monoid with Σ^* and we call a sequence $w = \sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_n \in \Sigma^*$ a word of length n.

Definition 2 (Deterministic Finite Automaton)
*A deterministic finite Automaton (**DFA**) is a tuple $A = (Q, \Sigma, \delta, q_\lambda, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_\lambda \in Q$ is a unique start state, $\delta : Q \times \Sigma \to Q \cup \{0\}$ is the transition function and $F \subseteq Q$ is the set of accepting states.*

This definition allows for partial **DFA**s, where a transition $\delta(q, \sigma) = 0$ implies a lack of knowledge about a transition. We will say in this case, the transition doesn't exist or is missing. To obtain a complete **DFA**, we can introduce an additional "sink" state 0, to which all missing transitions map and for which all outgoing transitions map to itself.

Given this definition, for each word $w = \sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_n \in \Sigma^*$, this implies a path $q_0, q_1, q_2, \ldots, q_n$ in A. We can construct this path in the following way: $q_0 = q_\lambda$ and $q_i = \delta(q_{i-1}, \sigma_{i-1})$. As a shorthand for this recursive definition, we will write $\delta(q_0, w) = q_n$. We will denote for such a word w, with q_w, the last state of this associated path and with $L_q = \{w \in \Sigma^* | v_w = q\}$ the set of all words, whose path ends in q. We say the automaton accepts w, if $q_w \in F$, otherwise, we say it rejects w. An automaton A defines a language $L_A = \bigcup_{q \in F} L_q$, the set of all words, it accepts.

Definition 3 (Edge-coloured graph)
Given a graph $G = (V, E)$, an edge-colouring of this graph is a mapping c from E to a set of colours Σ such that no two incident edges are mapped to the same colour. For directed graphs, we require that no two outgoing incident edges share the same colour. G can be a multi-graph, meaning there can be multiple edges between the same two nodes. Given such a mapping c, we call $G = (V, E, c)$ an edge-coloured graph.

We will denote the set of all edges incident to $v \in V$ as $\delta_v \subseteq E$, and with δ_v^-,δ_v^+ the out- and in-going edges respectively. We will use (u, v) to describe a directed edge from u to v of arbitrary colour.

We can convert an edge-coloured graph into a **DFA** by interpreting V as a set of states, choosing an initial state v_0 and a subset of V as F, as well as defining $\delta(u, x) = v$ if and only if $(u, v) \in E \wedge c((u, v)) = x$ and $\delta(u, x) = 0$ otherwise. This is well-defined, since no two incident edges can have the same colour. Note that this also allows for the reverse construction of an edge-coloured graph, given a **DFA**. In some places in this paper, we use graph algorithms, such as breadth-first search on a **DFA**. This conversion is implicitly done beforehand.

Definition 4 (Syntactic Monoid) *Given a language L, we define a congruence $\equiv_L$ on the free monoid Σ^*, where $u \equiv_L v \iff (\forall p, s \in \Sigma^* : p \cdot u \cdot s \in L \iff p \cdot v \cdot s \in L)$. This congruence gives us the quotient monoid $\Sigma(L) = \Sigma^* / \equiv_L$, which we call the syntactic monoid of L. Given a word $w \in \Sigma^*$ we denote with $[w]_L$ the congruence class $\{w' \in \Sigma^* | w \equiv_L w'\}$.*

Since this is a congruence $\forall u, v \in \Sigma^* : [u]_L \cdot [v]_L = [u \cdot v]_L$ holds by definition. We call the syntactic monoid finite if it contains a finite number of congruence classes. Note that a language L uniquely defines a syntactic monoid and conversely, a syntactic monoid recognises a language.

Definition 5 (Cayley Graph) *Note that the set $S = \{[\sigma]_L | \sigma \in \Sigma\}$ forms a generating set of the syntactic monoid. As such, we can create a directed edge-coloured graph, where we have a node for each congruence class in $\Sigma(L)$ and a colour for each element of Σ. When $e = ([u]_L, [w]_L) \in E$ of colour $c(e) = s \in \Sigma$, if and only if $[u]_L \cdot [s]_L = [w]_L$, we call it the right Cayley graph. Alternatively we call it the left Cayley Graph, when $e = ([u]_L, [w]_L) \in E$ of colour $c(e) = s \in \Sigma$, if and only if $[s]_L \cdot [u]_L = [w]_L$.*

Intuitively the right Cayley graph transitions between congruence classes by appending a symbol, and the left Cayley graph by prepending symbols. We will denote these graphs with $C_r = (V, E_r, c)$ and $C_l = (V, E_l, c)$ respectively. A syntactic monoid uniquely defines a Cayley graph.

The representation of the syntactic monoid with Cayley graphs is useful since we can extend them into **DFA**s by choosing $[\lambda]_L = q_\lambda$ and $[w]_L | w \in L$ as F and defining δ as shown previously. A **DFA** constructed in this way for the right Cayley graph recognises L.

3. State-merging algorithm

A popular way to learn **DFA**s from data employs state merging algorithms (Oncina and Garcia, 1992). They are an active area of research, and as such many varieties with optimisations for different tasks exist (Verwer and Hammerschmidt, 2022). You can see in Algorithm 1 a generic version of how they are structured. Which kind of states are considered mergeable can differ from algorithm to algorithm, however, there are some criteria, which essentially all of these state merging algorithms share, and which we will later use, to show the same algorithms can be used to learn syntactic monoids.

Algorithm 1: Structure of a generic state-merging algorithm for learning DFAs from traces

Input: A **DFA** $A = (Q, \Sigma, \delta, q_\lambda, Q^+)$ as well as a set Q^-, "representing" a dataset $D \subset \Sigma^*$, with $l : D \to \{0, 1\}$. Which means $\forall w \in D : l(w) = 0 \implies \delta(q_\lambda, w) \in Q^- \wedge l(w) = 1 \implies \delta(q_\lambda, w) \in Q^+$ and $Q^+ \cap Q^- = \emptyset$.
Output: An automaton A_f
while $\exists v, u \in Q : v$ *and* u *are mergeable* **do**
    Choose a mergeable pair of states $u, v \in Q$
    Merge u and v
end

Since Q changes from iteration to iteration due to merges, we will introduce some extra notation to better formulate these conditions. First, we will denote with Q_i the set of states present in iteration i. δ_i, Q_i^+ and Q_i^- will be defined analogously. Further, we introduce

the function $r_i : Q_i \to Q_{i+1}$, which maps a state $q \in Q_i$ to its representative $x \in Q_{i+1}$. In other words, if for $u, v \in Q_i$ $r_i(u) = r_i(v)$, then u and v were merged in iteration i of the algorithm. A **merge** in iteration i between u and v then consists of setting $r_i(u) = v$. We will denote with $q_{\lambda i}$ the state $r_i(q_{\lambda i-1})$, where $q_{\lambda 0} = q_\lambda$.

Definition 6 (State-merging properties) *We will only consider two states $u, v \in V$ mergeable, w.r.t. algorithm 1, if the merge, and all subsequent merges caused by it, fulfil the following properties: They are **deterministic**, **consistent** and **contiguous**.*

- ***deterministic**:*
  $\forall i : \forall u, v \in Q_i$ *with* $r_i(u) = r_i(v) : \forall \sigma \in \Sigma : \delta_i(u, \sigma) = x \wedge \delta_i(v, \sigma) = y$ *with* $x, y \neq 0$
  $\implies r_i(x) = r_i(y)$
  In other words, if two nodes states merged and both have a successor reached with the same symbol, their successor also needs to be merged in the same iteration.

- ***consistent**:*
  $\forall i : \forall u \in Q_i^+ : (r_i(u) \in Q_{i+1}^+)$
  $\forall i : \forall v \in Q_i^- : (r_i(v) \in Q_{i+1}^-)$
  $\forall i : \forall u \in Q_i^+, v \in Q_i^- : (r_i(u) \neq r_i(v))$
  In other words, if one of the states, that is merged was an accepting/rejecting state, the new representative is also an accepting/rejecting state and accepting are never merged with rejecting states.

- ***contiguous**:*
  $\forall i : \forall \sigma \in \Sigma :$
  $(\exists u \in Q_i : \delta_i(u, \sigma) = v$ *with* $v \neq 0 \iff \delta_{i+1}(r_i(u), \sigma) = r_i(v)$ *with* $r_i(v) \neq 0)$
  In other words, if a transition is present in iteration i it still needs to be present between the representatives of its states in iteration $i+1$ and conversely, if a transition exists in iteration $i+1$ it must have already existed in iteration i for at least one pair of states that got merged into the states of this transition.

With these properties, we can show a very useful result for Algorithm 1:

Lemma 7

*For each iteration i of Algorithm 1, $A_i = (Q_i, \Sigma, \delta_i, q_{\lambda i}, Q_i^+)$ is a **DFA** with $L_{A_i} \subseteq L_{A_{i+1}}$ and $L_{A_{i+1}} \cap \bigcup_{q \in Q_i^-} L_q = \emptyset$.*

Proof

Since $Q_0^+ \cap Q_0^- = \emptyset$ the lemma holds trivially for $i = 0$. Now we will perform an induction by considering the merges that happen between iteration i and $i+1$. To prove that $L_{A_i} \subseteq L_{A_{i+1}}$ and $L_{A_{i+1}} \cap \bigcup_{v \in V_i^-} L_v = \emptyset$, for any $q \in Q_i$ we will take an arbitrary word $w \in L_q$ and show $w \in L_{r_i(q)}$. Since merges are **consistent** $q \in Q_i^+ \implies r_i(q) \in Q_{i+1}^+$, and equivalently for Q^-. As such, this will be sufficient to prove our claim. Recall that for each **DFA** there exists an implied path for each word w, consisting of a series of states $q_{\lambda i}, q_1, \ldots, q_n$. Since merges are **contiguous**, we can map such a path in A_i, which ends in $q_n = q_w$ to a path in A_{i+1}. Namely, the path $r_i(q_{\lambda i}), r_i(q_1), \ldots, r_i(q_n)$. Furthermore, since merges are **deterministic** this mapping is unique.

■

A well-known result on regular languages is that **SMF** and **DFA** are equally powerful for what kind of languages they can represent (McNaughton and Papert, 1968). In fact, given an **SMF**, we can easily construct a **DFA** that recognises the same language and is smaller by simply taking the right Cayley graph. And this **DFA** normally can be made even smaller by minimising its size. Although it is larger and language-equivalent, an **SMF** has one key advantage over a **DFA**: it constructs words starting at any index. This could solve a well-known weakness of state-merging algorithms for DFAs, which is that they learn from trace prefixes, making it harder to learn patterns that appear at later indexes in traces. An **SMF** state-merging algorithm can learn all such patterns at once. Hence even when the prefix patterns are hard to learn, such interfix patterns can still be correctly picked up.

3.1. Interfix-Graph

Data for state machine learning normally consists of an alphabet Σ, a set of words over this alphabet $D \subseteq \Sigma^*$, and for each word $w \in D$ a frequency function $f : D \to \mathbb{N}$, that signifies how often a word occurs in the data, as well as an indicator function $l : D \to \{0, 1\}$, that maps to 1 if w is in the language L and 0 otherwise. The goal is to learn a structure that recognises exactly L. The last missing piece for learning an **SMF** from data is how to encode such data into an input of algorithm 1.

Traditionally this data is encoded into a prefix tree (Oncina and Garcia, 1992). A **DFA** is then obtained by passing the prefix tree to a state-merging algorithm satisfying the conditions in definition 6.

Definition 8 (Prefix Tree) *Given an alphabet Σ and a set of data $D \subset \Sigma^*$, as well as a membership indicator $l : D \to \{0, 1\}$ a prefix tree is a **DFA** $A_{pt}(D) = (Q, \Sigma, \delta, q_\lambda, Q^+, Q^-)$ satisfying the following conditions:*

- $\forall q \in Q : |L_q| = 1$
  In other words, each state in the prefix tree is reachable by exactly one word in Σ^*.
- $\forall w = z_1 \cdot z_2 \ \ldots \ \cdot z_n \in D : \forall i \in [1, n] : \exists q \in Q : L_q = \{\lambda \cdot \ \ldots \ \cdot z_i\}$
  In other words, every possible prefix in D is represented by a state in the prefix tree.
- *When interpreted as an edge-coloured graph, the prefix tree is a directed tree, with v_λ as its root.*
- $\forall w \in D : (l(w) = 1 \implies q_w \in Q^+) \wedge (l(w) = 0 \implies q_w \in Q^-)$
  In other words, states representing positive samples are accepting, and ones representing negative samples rejecting.

Note that this definition points to a unique structure, which can be easily constructed by creating a state for each prefix present in D and the defining δ as $\delta(q_u, a) = q_v$ if there exists a word $w = u \cdot a \cdot r$ in D for some $r \in \Sigma^*$ and $\delta(q_u, a) = 0$ otherwise. Finally, Q^+ and Q^- are constructed according to the last condition of the definition.

The prefix tree is a great input for state merging, as it is a **DFA**, which recognises exactly $L = \{w \in D | l(w) = 1\}$. From Lemma 7 we therefore obtain, that state-merging will

always return a language, which has L as a subset. Further, each possible prefix of words in D is represented by a unique state. Thinking from the perspective of a **SMF**, we can think of the prefix-tree as a right Cayley graph. So in order to generalise this concept for learning **SMF**, we also need to encode the left Cayley graph. This leads us to something we call the interfix-graph (**IG**):

Essentially, we design the **IG** to have exactly one node for each interfix in our sample. To define the relations we introduce some extra notation. We define additional colours Σ_{pp} with $|\Sigma_{pp}| = |\Sigma|$ and $\Sigma_{pp} \cap \Sigma = \emptyset$, allowing for a bijective function $m : \Sigma \to \Sigma_{pp}$, and m^{-1} its inverse. The idea is that two states representing interfixes u, v have a relation $\delta_{right}(q_u, a) = q_v$ if v can be obtained from u by *appending* a. But they have a relation $\delta_{left}(q_u, m(a)) = q_v$ instead, if v can be obtained from u by *prepending* a. Therefore, δ_{right} represents right actions and δ_{left} left actions.

Definition 9 (Interfix-Graph) *Given an alphabet Σ and a set of data $D \subset \Sigma^*$, as well as a membership indicator $l : D \to \{0, 1\}$, we define with $D_I = \bigcup_{w \in D} \{\sigma_i \cdot \sigma_{i+1} \dots \sigma_n | \sigma_1 \dots \sigma_n = w, \forall i \in [1, n]\}$ the extended sample, which includes additionally all suffixes present in D. From this, we give a constructive definition of the interfix-graph. First we build a prefix tree $A_{pt}(D_I) = (Q, \Sigma, \delta_{right}, q_\lambda, Q^+, Q^-)$. A_{pt} now has a separate state for each possible interfix in D, instead of each prefix.*

We define a new function $\delta_{left} : Q \times \Sigma_{pp} \to Q \cup \{0\}$:

- $\forall q_w, q_a \in Q : (\exists t \in \Sigma : t \cdot w = a \implies \delta_{left}(q_w, m(t)) = q_a)$
- $\forall q_w, q_a \in Q : (\nexists t \in \Sigma : t \cdot w = a \implies \delta_{left}(q_w, m(t)) = 0)$

Finally we define $\delta : Q \times (\Sigma \cup \Sigma_{pp}) \to Q \cup \{0\}$ as

$$\delta(q, a) = \begin{cases} \delta_{right}(q, a) & a \in \Sigma \\ \delta_{left}(q, a) & a \in \Sigma_{pp} \end{cases}$$

Together this forms the Interfix-Graph $I(D) = (Q, \Sigma, \delta, v_\lambda, Q^+, Q^-)$.

Figure 1 shows a simple prefix tree and interfix-graph to illustrate definition 9. Having defined this structure we will now show in the following theorem that a state-merging algorithm initialised with this structure will return a **DFA** which we can easily split into the left and right Cayley graph of a syntactic monoid.

Theorem 10 (Learning a Syntactic Monoid)

A state-merging algorithm, that fulfills the properties in Defintion 6, when initalised with an Interfix-Graph will terminate with $A_f = (Q_f, \Sigma \cup \Sigma_{pp}, \delta_f, q_{\lambda f}, Q_f^+, Q_f^-)$. We will use δ_{right} and δ_{left} as in definition 9. A_f satisfies:

- *$A_{right} = (Q_f, \Sigma, \delta_{right}, q_{\lambda f}, Q_f^+)$ is an automaton that recognises a language L such that $\{w \in D | l(w) = 1\} \subseteq L$ and $L \cap \{w \in D | l(w) = 0\} = \emptyset$.*

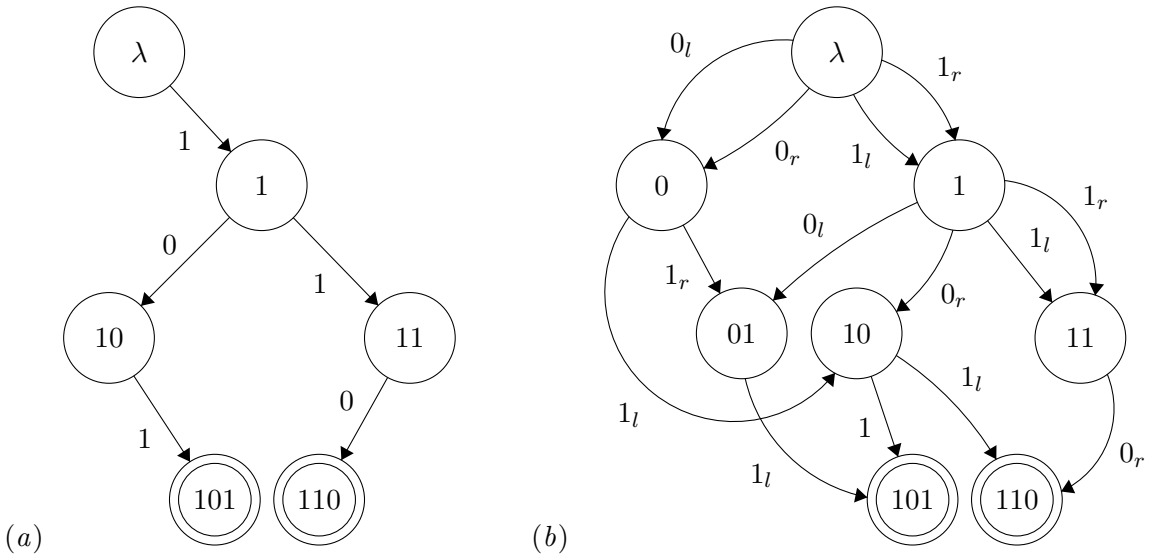

Figure 1: Prefix tree (a) and interfix-graph (b) for data consisting of the two positive traces 101 and 110. For the **IG** connections with subscript r belong to δ_{right} and connections with subscript l belong to δ_{left}. States in Q^+ are represented by a double circle.

- *For all $q \in Q_f$ except for at most one of them, L_q is a congruence class of the syntactic congruence $\equiv_L$ and there either exists a bijective mapping between Q_f and the set of congruence classes or between $Q_f \cup \{0\}$ and the set of congruence classes.*

- *Using the construction under definition 3, $A_{right} = (Q_f, \Sigma, \delta_{right}, q_{\lambda f}, Q_f^+)$ converts to the right Cayley graph of the syntactic monoid that recognises L and $A_{left} = (Q_f, \Sigma, \delta_{left}, q_{\lambda f}, Q_f^+)$ to the left Cayley graph.*

Proof

- The first property we get easily from Lemma 7, since $A_{right0} = (Q_0, \Sigma, \delta_{right0}, q_{\lambda 0}, Q_0^+)$ is the prefix tree, which recognises exactly D. Since no words containing characters from Σ_{pp} can be part of the language recognised by A_{right} it is easy to see, that Lemma 7 will remain valid on this sub-automaton.

- The second property is proven using that the state merging was **deterministic**, **consistent** and **contiguous**. Since the algorithm terminates, no further merges fulfilling all three properties existed. Now assume $\bigcup_{q \in Q_f}\{L_q\}$ is not a subset of congruence classes of L. This implies, there exists $v, u \in Q_f : \exists w_v \in L_v : \exists w_u \in L_u : w_v \equiv_L w_u$. This implies $\forall p, s \in \Sigma^* : pw_v s \in L \iff pw_u s \in L$. Since we assume no further merges are possible, when merging u and v, there must exist a merge caused due to the **deterministic** property that is not **consistent**. The **deterministic** property

lets us construct a sequence of characters from $d \in (\Sigma \cup \Sigma_{pp})^*$, such that $\delta_f(u, d) = u'$ and $\delta_f(v, d) = v'$ for some $u', v' \in Q_f$. Under our assumption, there must exist a d, where merging u' and v' is not **consistent**. As such $L_{u'} \subseteq L \iff L_{v'} \cap L = \emptyset$, since one of them being in Q^+ implies the other is in Q^-.

We will now split d into the sequence containing only colours in Σ, d_{right}, and the one containing only colours in Σ_{pp}, d_{left}. Since all merges are **contiguous** and in the **IG** an edge with a colour $t \in \Sigma$ implies $\exists w, q \in D_I : w \cdot t = q$, an edge of colour t between two nodes u, v in A_f implies $\exists w \in L_u : \exists q \in L_v : w \cdot t = q$. Equivalently an edge of colour $m(t) \in \Sigma_{pp}$ between $u, v \in V_f$ implies $\exists w \in L_u : \exists q \in L_v : t \cdot w = q$. As such d_{right} forms a suffix, when following its edges, conversely d_{left} implies a prefix (the reverse sequence of d_{left}), which we will name d'_{left}. Put together this implies, that given $w_u \in L_u$ and $w_v \in L_v$, that $d'_{left} \cdot w_u \cdot d_{right} \in Q_f^+$ and $d'_{left} \cdot w_v \cdot d_{right} \in Q_f^-$, which is a contradiction to our assumption $\forall p, s \in \Sigma^* : pw_v s \in L \iff pw_u s \in L$. Thus L_u and L_v are congruence classes of L.

With this, we have shown, that two words that map to different states with A_f can not belong to the same congruence class of L. To complete the proof, we also need to show, that each word in Σ^* maps to at least one such set and as such these sets form the complete congruence classes.

To show this, we need to handle an edge case, which can also lead to at most one of these sets to be only a subset of a congruence class. There exist transitions of the form $\delta_f(q, c) = 0$. From our definition of the **deterministic** property, these transitions are excluded. They can not lead to inconsistencies. We will define $L_0 = \{w \in \Sigma^* | \delta_f(q_\lambda, w) = 0\}$, the set of all words, which encounters such an unknown transition. None of these words are in L, since 0 can not be in Q_f^+, and since 0 is a "sink", for any $w \in L_0 : \forall l, r \in \Sigma^* : l \cdot w \cdot r \in L_0$. If there exists a state $q_s \in Q_f$ which is also a "sink", so $\forall c \in \Sigma \cup \Sigma_{pp} : \delta_f(q_s, c) = q_s$, then it follows that for all $v \in L_0$ and $u \in L_{q_s}$ $v \equiv_L u$. In other words $L_0 \cup L_{q_s}$ forms a congruence class of L. If no such state exists, L_0 is a congruence class of L. Since $\bigcup_{q \in Q_f} L_q \cup L_0 = \Sigma^*$ we have proven our claim since the mapping is directly implied.

- Since $\Sigma / \equiv_L$ forms a generating set the second property immediately implies the third. That is under the assumption, that the graphs were completed to also include the state 0, or if another sink state q_s existed, all unknown transitions were converted into edges to that state.

■

Theorem 10 shows that we learn a structure, which we can interpret as the syntactic monoid of a language L. However, we still need to show, that we learn the correct language from which D is sampled in the limit.

Theorem 11 *Given an Interfix-Graph as an input, a state-merging algorithm that is **deterministic**, **consistent** and **contiguous** will learn a regular language L in the limit, given an informant on L (Gold, 1967).*

The proof for theorem 11 is provided in Appendix B.

3.1.1. Encoding of Frequencies

With Theorem 10 we have shown, that we can use existing algorithms to learn **SMF** if we use an **IG** as input. However, this algorithm, same as with **DFA**, will just learn **SMF** that recognises Σ^* when given only positive samples. This is due to all merges being **consistent** by default. Therefore, when learning **DFA** from such data current methods replace the check for consistency with one for similarity. Many different similarity measures are in use, but most require some probability distribution over the possible transitions from a node. Without defining such probabilities, it is not possible to apply such algorithms and learn from only positive samples. For a prefix tree, these probabilities are easily defined from the data by simply recording transition frequencies. This is possible since in the prefix tree for each $w \in D$ there is a unique path from v_λ to v_w (Carrasco and Oncina, 1994). Accordingly, for each word, we can just increase a counter for each node and edge on this path by 1, which then defines a probability distribution in each node when normalised. This distribution also perfectly reflects the distribution of the frequencies of words in the input.

For an **IG**, we can not directly copy this approach, since the paths from v_λ to v_w are not unique. This is due to it being possible to construct a word with combinations of pre- and appending operations in many different ways. Since we still want to use frequency information for heuristics, as well as be able to learn from only positive samples, we adapted the method for **DFA** in the intuitive way as seen in algorithm 2.

Algorithm 2: Algorithm to encode frequency of a trace

Input: An Interfix-Graph $G_I = (V, E \cup E_{left}, c, v_\lambda)$, as well as a trace $w = z_0...z_n \in \Sigma^*$
Output: A frequency function f^w for nodes and f_E^w for edges.
Identify subgraph $G \subseteq G_I$ of all nodes w could visit on a path from v_λ to v_w ;
Sum of incoming frequencies for $f^w(v_\lambda) \leftarrow 1$;
Perform Breadth First Search **BFS** on G starting at v_λ ;
foreach *v visited during the* ***BFS*** **do**
    $f^w(v) \leftarrow$ sum of frequencies from incoming edges ;
    foreach $e \in \delta_v$ **do**
        $f_E^w(e) \leftarrow \frac{f(v)}{|\delta_v|}$;
    end
end

Essentially Algorithm 2 identifies all possible paths a trace could take and then each time such a path branches assumes all branches are equally likely. Assuming this equal probability of branches, the values $f^w(v)$ represent the probability that a randomly chosen path among all possible paths w could take through the model would pass through v. Analogously $f_E^w(e)$ represents the probability, that said path would use edge e. As such this approach assigns frequencies between 0 and 1 to all nodes and edges that appear on paths w could take. The frequencies of v_λ and v_w are 1 with this method since all paths begin and end in these nodes. Figure 2 illustrates how the algorithm works on an example instance.

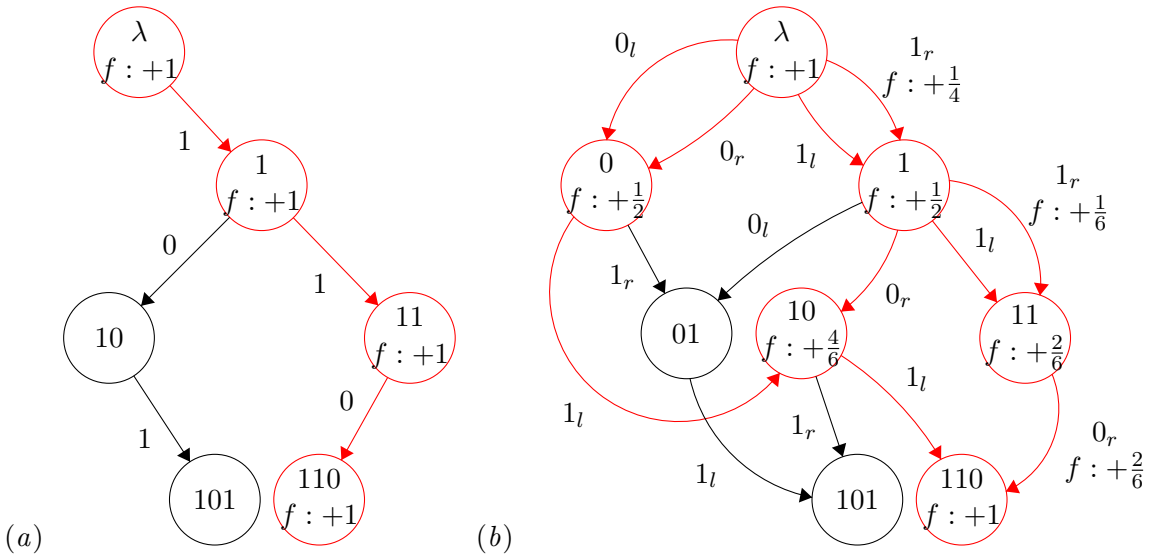

Figure 2: Example of frequencies after seeing the traces 110 on the graphs generated in Figure 1. The subgraph traversed by Algorithm 2 is coloured in red. The frequency adjustments for the rightmost path are also shown.

To obtain the frequencies for the entire sample D, we sum up the results of each trace:

$$\forall v \in V : f(v) = \sum_{w \in D} f^w(v)$$
$$\forall e \in E : f_E^w(e) = \sum_{w \in D} f_E^w(e)$$

We will now assume the viewpoint of a probabilistic generating model as given by (Carrasco and Oncina, 1994). Given frequencies in the way we defined, we can estimate the probability, that a word ends at node v_w, $p_t : \Sigma^* \to [0,1]$ with $p_t(w) = \frac{\sum_{e \in \delta(v_w)} f_E(e)}{f(v_w)}$. Further, the probability $p_a : \Sigma^* \to [0,1]$ that for a word $w = u \cdot w_i \cdot z$, where $u, z \in \Sigma$ and $w_i \in \Sigma^*$ the model will generate a string up to w and thus arrives in the state v_w, can be computed using the set of equations in 1.

$$\begin{aligned}
p_a(\lambda) &= 1 \\
p_a(u) &= p_a(\lambda) * \left(\frac{f_E((v_\lambda, u, v_u))}{f(v_\lambda)} + \frac{f_E((v_\lambda, m(u), v_u))}{f(v_\lambda)}\right) \\
p_a(u \cdot z) &= p_a(u) * \frac{f_E((v_u, z, v_{uz}))}{f(v_u)} + p_a(z) * \frac{f_E((v_z, m(u), v_{uz}))}{f(v_z)} \\
p_a(u \cdot w_i \cdot z) &= p_a(uw_i) * \frac{f_E((v_{uw_i}, z, v_{uw_iz}))}{f(v_{uw_i})} + p_a(w_iz) * \frac{f_E((v_{w_iz}, m(u), v_{uw_iz}))}{f(v_{w_iz})}
\end{aligned} \tag{1}$$

This can be efficiently computed using dynamic programming. Given p_t and p_a we estimate the probability that a model produces a word w as $p(w) = p_t(w) * p_a(w)$.

4. Results

4.1. Experimental setup

To test how **SMF** perform compared to **DFA**, we implemented a traditional state-merging algorithm. We then used the same algorithm to learn **SMF** and **DFA** for several datasets and compared their performance. We used the same parameters for both **SMF** and **DFA**. These results are not meant to challenge state-of-the-art results on these datasets but give insight into the relative performance of **DFA** and **SMF**. Theorem 10 implies that many of the techniques to achieve state-of-the-art results, such as heuristics or similarity measures (Verwer and Hammerschmidt, 2022), are also possible with **IG**s as inputs.

4.1.1. Algorithm

The state-merging procedure we implemented is the blue-fringe state-merging algorithm (Lang et al., 1998). It keeps a core of "red" states, where in the beginning only q_λ is red, and all their non-"red" successors as "blue" states. Only merges between red and blue states are considered in each iteration. If no merge is possible for a blue state, it is recoloured as red.

As a heuristic, for which merge to perform in each iteration, if multiple are possible, we implemented four approaches. First, the states with the highest frequency are merged first, which we considered as a baseline, second we implemented the well-known **EDSM** heuristic (Lang et al., 1998), third we ran an adjusted version of **EDSM**, which also considers states to have the same label if all their successors have the same label (Future agreement, **FA**), and lastly the also popular method of performing the merge, that merges the highest number of states during determinisation (Verwer and Hammerschmidt, 2022).

For the state-merging two details needed to be adjusted from the standard implementation, in order to work with **IG**s as input. First, if two red states are considered for a merge, the merge is rejected as inconsistent, second, if two states, which are already marked for being merged in the same iteration are considered, the merge is classified as consistent. Both are cases which are not possible when initialising with a prefix tree. If not handled, they lead to infinite loops, however, when handled, the algorithm terminates as expected (For a running time analysis see Theorem 12 in the appendix). For the similarity check, when working with only positive data, we used the classical Alergia check (Carrasco and Oncina, 1994).

All results were generated on an Intel i5 2.6GHz processor. The full code, data and results, can be found at https://github.com/SimonDieck/Syntactic_Monoid_Passive_Learner.

4.1.2. Data

Two datasets were used for the experiments. For traditional data with positive and negative samples, we used instances $1-25$ from the 2010 *stamina* challenge (Walkinshaw et al., 2013). We only had access to the training sets, since the original competition site provides only unlabelled test sets and no mechanism to test accuracy on these files anymore. Hence, we re-split those training files on unique traces with an 80/20 ratio. Balanced accuracy was chosen as the evaluation metric.

For a dataset with only positive traces, we used the first 20 instances of the 2014 Pautomac challenge (Verwer et al., 2014). For evaluation, we used the perplexity measure proposed in this challenge.

4.2. Results on traditional data

The **SMF** performed noticeably worse than the **DFA** on the *stamina* dataset. The **SMF** outperformed the **DFA** on only 2 out of 25 instances and had an average balanced accuracy of only 53.8 as opposed to the average balanced accuracy of 74.1 by the **DFA**. Essentially the **SMF** only learned a grammar that generalised well to unseen data on instance 22 and only outperformed the **DFA** on instances where it performed very poorly, to begin with. As for the run time, it was below 1 second for the learning of all **DFA**, while the time taken to learn the **SMF** varied significantly from 1.5 seconds to 11.7 hours. The average model size of the **DFA** was 91, while it was $15,794$ for the **SMF**. Full results in Appendix C.

To understand this poor performance we analysed instance 1 of stamina closely. The structure of the language of this instance, L_1, is strongly prefix dependent. A prefix of 111 guarantees a word is rejected, while the behaviour with prefix 00 can be summarised by the automaton in Appendix D. A prefix of 01 causes more complex behaviour. The issue for the **SMF** arises from the fact that behaviour in one of these subtrees cannot be ignored in the others. For example, the simple sink state containing 111 would need to be extended as shown in Figure 3 since 00 could be prepended at any time. Moreover, it also needs to be extended to cover the behaviour of the subtree starting with 01, which is too complex to portray in this paper. This also exemplifies a blowup in states one can expect when going from an automaton to a syntactic monoid. In the worst case this blowup can actually be exponential as shown by Holzer and König (2004).

This causes two issues when learning the **SMF**. To make this clearer, assume an **IG** with almost complete information along our example in Figure 3. If the edges with colour 00 exist in the graph the states containing 1110 and 1111 cannot be merged since determinisation would discover an inconsistent merge in $00 \cdot 11110$ and $00 \cdot 1111$. On the other hand, merging 111 and 1110 would also ensure $00 \cdot 111$ and $00 \cdot 1110$ are merged due to determinisation. This ensures that both subtrees have the same mirrored structure. A key reason for the two issues mentioned earlier is that these two determinisation mechanisms work incorrectly due to missing information. Say we do not know if $00 \cdot 1111$ is accepting, then the merge between 1111 and 11110 will not result in an inconsistency and determinisation will actually propagate this incorrect merge, preventing e.g. that in a future merge, 11110 is merged with 000. The other issue arises when edges are missing. Say there is no outgoing edge of colour 0 for 111, then the determinisation that would ensure the mirrored subtrees does not happen, which leads to incorrect determinisation in future steps.

4.3. Results on data with only positive samples

For the *Pautomac* dataset, the **SMF** had much more competitive results. While it also only outperformed the **DFA** on 5 out of 16 instances, except for 2 instances (4 and 7), the perplexity was within 25% of the **DFA** complexity. On instances 10 and 20 the **SMF** even outperformed the **DFA** by more than 20%. Furthermore, when analysing the results more closely, the **SMF** tended to perform worse on traces with high probability and better on

traces with low probability. However, the **SMF** required very long running times on the *Pautomac* dataset, hitting an early timeout on 3 out of 20 instances, because it did not reduce the initial number of states by 10% within 1 hour. Full results in Appendix C.

5. Discussion

Being able to learn **SMF** efficiently with already existing algorithms could be of significant benefit to the field of learning grammars for regular languages. It allows us to learn a different kind of grammar, which could be of advantage for several problems. For example, all of Green's relations can be checked for a grammar learned in this way since they correspond to the strongly connected components in the Cayley graphs (Clifford and Preston, 1961). Further, the probabilistic version of the **SMF** could be of benefit for some applications. Performing poorly on low probability traces has been a problem with **DFA** for several applications. If further experiments on other datasets show that the **SMF** performs consistently well on such traces, they could be used in a complementary fashion for such problems. The poor performance on traditional data will require more research to address. For example, heuristics or algorithms which aim to merge more than two states in one step could tackle the issue of properly learning mirrored substructures in multiple parts of the **SMF** at once. Another area that needs to be addressed is the lower information density in the **IG** since a higher density of labelled states will make incorrect merges less frequent. While there are significantly more states in an **IG** than in a prefix tree, the number of states labelled as rejecting or accepting is the same between them.

Overall, we believe that the results presented in this paper are a good starting point for further research. The algorithms and heuristics instrumented in this work to learn **SMF** are all optimised for learning **DFA**. Future work can now be used to optimise them for learning **SMF** instead. Results in this paper also provide a theoretical basis to design other input structures for state-merging algorithms, that future work could use to learn entirely novel grammar structures that might be able to combine the advantages of **SMF** and **DFA**.

One limitation, however, is the increase in model size for the **SMF** compared to **DFA**. It is known that the **SMF** can be exponentially larger in the worst case and we observed quadratic to cubic differences in our experiments. As such **DFA** will be the better option if higher interpretability is desired. Nevertheless, there might be problems, where the left relations in the **SMF** give additional insight not found in a **DFA**.

Learning an **SMF** instead of a **DFA** will likely be a trade-off that needs to be considered between different use cases. The results in this paper should be taken as a good foundation to further explore and research this trade-off.

References

Bernhard K Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. Model learning and model-based testing. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, pages 74–100. Springer, 2018.

Shahbaz Ali, Hailong Sun, and Yongwang Zhao. Model learning: a survey of foundations, tools and applications. *Frontiers of Computer Science*, 15:1–22, 2021.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

Rafael C Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In *Grammatical Inference and Applications: Second International Colloquium, ICGI-94 Alicante, Spain, September 21–23, 1994 Proceedings 2*, pages 139–152. Springer, 1994.

Alexander Clark. Learning context free grammars with the syntactic concept lattice. In *Grammatical Inference: Theoretical Results and Applications: 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings 10*, pages 38–51. Springer, 2010a.

Alexander Clark. Three learnable models for the description of language. In *LATA*, pages 16–31. Springer, 2010b.

Alexander Clark. Canonical context-free grammars and strong learning: two approaches. In *Proceedings of the 14th Meeting on the Mathematics of Language (MOL 2015)*, pages 99–111, 2015.

Alexander Clark and Rémi Eyraud. Polynomial identification in the limit of substitutable context-free languages. *Journal of Machine Learning Research*, 8(8), 2007.

Alfred Hoblitzelle Clifford and GB Preston. *The Algebraic Theory of Semigroups, Volume I*, volume 7. American Mathematical Soc., 1961.

Henning Fernau. Algorithms for learning regular expressions. In *Algorithmic Learning Theory: 16th International Conference, ALT 2005, Singapore, October 8-11, 2005. Proceedings 16*, pages 297–311. Springer, 2005.

E Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(67)91165-5.

Markus Holzer and Barbara König. On deterministic finite automata and syntactic monoid size. *Theoretical Computer Science*, 327(3):319–347, 2004.

Gerhard Jäger and James Rogers. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598):1956–1970, 2012.

Mati Kilp, Ulrich Knauer, and Alexander V. Mikhalev. *Monoids, Acts and Categories*. De Gruyter, Berlin, New York, 2000. ISBN 9783110812909. doi: 10.1515/9783110812909.

Bernhard H Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial optimization*, volume 1. Springer, 2011.

Kevin Lang, Barak A Pearlmutter, and Rodney Price. Results of the abbadingo one DFA learning competition and a new evidence driven state merging algorithm. In *Fourth International Colloquium on Grammatical Inference (ICGI-98)*, volume 98, 1998.

Petr Matoušek, Vojtěch Havlena, and Lukáš Holík. Efficient modelling of ICS communication for anomaly detection using probabilistic automata. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 81–89. IEEE, 2021.

Robert McNaughton and Seymour Papert. The syntactic monoid of a regular event. *Algebraic Theory of Machines, Languages, and Semigroups*, pages 297–312, 1968.

José Oncina and Pedro Garcia. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*, pages 99–108. World Scientific, 1992.

Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Inf. Control.*, 8(2):190–194, 1965.

Frank Stephan and Yuri Ventsov. Learning algebraic structures from text. *Theoretical Computer Science*, 268(2):221–273, 2001.

Jacques Stern. Complexity of some problems from the theory of automata. *INFO. CONTROL.*, 66(3):163–176, 1985.

Sicco Verwer and Christian Hammerschmidt. Flexfringe: Modeling software behavior by learning probabilistic automata. *arXiv preprint arXiv:2203.16331*, 2022.

Sicco Verwer, Rémi Eyraud, and Colin De La Higuera. Pautomac: a probabilistic automata and hidden markov models learning competition. *Machine learning*, 96:129–154, 2014.

Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. Stamina: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering*, 18(4):791–824, 2013.

Shunjie Wang. *Evaluating Transformer's Ability to Learn Mildly Context-Sensitive Languages*. University of Washington, 2021.

Wojciech Wieczorek and Olgierd Unold. Use of a novel grammatical inference approach in classification of amyloidogenic hexapeptides. *Computational and mathematical methods in medicine*, 2016, 2016.

Appendix A. Running Time

Theorem 12 (Blue-Fringe running time) *The blue fringe algorithm initialised with an Interfix-Graph terminates in $O(n^3 * \alpha(n) * |\Sigma|)$, where n is the number of states in the Interfix-Graph, or conversely, the number of unique interfixes in the input sample, α is the inverse of the Ackerman function and hence almost constant for most practical inputs and $|\Sigma|$ is the size of the alphabet.*

Proof The bound follows directly from two simple observations. The first one is the observation, that it takes at most $O(n * \alpha(n) * |\Sigma|)$ steps to check if a merge is possible. During each individual merge the label of the states needs to be determined and compared and for each outgoing edge, a subsequent merge might be called due to determinisation. Checking the labels can be done in $\alpha(n)$, if merges are maintained with a Uninon-Find data structure (Korte et al., 2011), while there can be at most $|\Sigma|$ edges. Determinisation can cause at most n such merge attempts, since during each attempt either an inconsistency is found and the attempt terminates, or two states are marked to be merged. Since there exist at most n states, all states would be merged into one after at most n such attempts.

The second observation is that in each outer iteration, we select one blue state and attempt to merge it with all red states. If no merge is possible the blue state is changed into a red state. Hence in the first iteration one merge is attempted, since there is only one red state, in the second at most two, and so on. Since there are only n states in total and in each successful merge the number of total states is reduced by at least one, there can be at most n such iterations. Hence we can limit the total number of merge attempts with $O(\sum_{i=1}^{n} i) = O(n^2)$.

Combining these two observations results in the claimed running time. ■

Appendix B. Learning in the Limit

This section provides the proof for theorem 11 that was omitted in the main body of the paper.

Proof We will denote with $C_r^L = (V, E_r, c_r)$, $C_l^L = (V, E_l, c_l)$ the left and right Cayley Graph of the **SMF** of L. Accordingly, V can be interpreted as the set of all congruence classes of L. As such, when we write for $v \in V$ $[v]_L$ we mean $[w]_L$, where $w \in L_v$.

A state-merging algorithm will perform all merges that are possible, in some order, given an input structure, which is an incomplete edge-coloured graph. Further, since L is regular, V is finite and there exists an $n \in \mathbb{N}$, such that for every two states $v, w \in V$ with $v \neq w$, for all words $z_v \in L_v, z_w \in L_w$ there exists a pair of strings $l, r \in \Sigma^*$, s.t. $l \cdot z_v \cdot r \in L \iff l \cdot z_w \cdot r \notin L$, with $|l \cdot z_v \cdot r| \leq n$ and $|l \cdot z_w \cdot r| \leq n$. This is due to the pumping lemma.

Now assume an informant has shown us all possible samples of length smaller than n. Then when attempting to merge two states x, y, with $[x]_L \neq [y]_L$ there must exist a path from both x, as well as y, s.t. the edges from E_{left} form the inverse of l and the edges in E form r, and for the two nodes respectively at the end of this path p_x, p_y, $l(p_x) \neq l(p_y)$ must hold. Accordingly due to the **deterministic** and **consistent** properties of the algorithm the states can't be merged. Further if $[x]_L = [y]_L$ holds such a path can not exist, since the **contiguous** property of the algorithm would otherwise imply the existence of and l, r, s.t. $lw_xr \in L \iff lw_yr \notin L$ for a $w_x \in L_x, w_y \in L_y$.

After being shown all possible samples of length smaller then n the algorithm will therefore always terminate with $A_f = (V_f, E_{right} \cup E_{left}, c_f^{'}, v_\lambda, V_f^+, V_f^-)$, where $(V_f, E_{right}, c_f^{'}) = C_r^L$ and $(V_f, E_{left}, m^{-1} \cdot c_f^{'}) = C_l^L$. Correctly identifying L in the limit.

E_{left} and E_{right} are guaranteed to be complete since all samples of length smaller than n were encoded in the interfix graph and V_f is minimal since all possible merges are attempted. ■

Appendix C. Results

FileID	Model	Best heuristic	States in input	States in output	In/Out ratio	Time	Balanced Acc
1	DFA	Baseline	8711	49	177.8	0.01s	**0.99**
1	FSM	States merged	43330	11347	3.8	271.98s	0.52
2	DFA	EDSM	25214	68	370.8	0.02s	**0.99**
2	FSM	Baseline	179991	45310	4.0	8299.39s	0.54
3	DFA	Baseline	20593	73	282.1	0.01s	**0.99**
3	FSM	States merged	121745	30625	4.0	3178.88s	0.53
4	DFA	States merged	12335	55	224.3	0.02s	**0.99**
4	FSM	FA	72535	18974	3.8	1076.84s	0.49
5	DFA	Baseline	11883	63	188.6	0.01s	**0.99**
5	FSM	EDSM	74992	19040	3.9	1458.56s	0.50
6	DFA	EDSM	10354	222	46.6	0.11s	**0.55**
6	FSM	States merged	69173	17372	4.0	674.81s	0.52
7	DFA	States merged	7588	156	48.6	0.06s	**0.71**
7	FSM	EDSM	54153	12409	4.4	430.37s	0.51
8	DFA	FA	3039	112	27.1	0.02s	**0.64**
8	FSM	FA	9775	2540	3.8	8.55s	0.54
9	DFA	EDSM	16169	74	218.5	0.01s	**0.95**
9	FSM	States merged	108881	24136	4.5	1470.35s	0.54
10	DFA	EDSM	6034	92	65.6	0.01s	**0.86**
10	FSM	EDSM	26078	6164	4.2	62.79s	0.52
11	DFA	States merged	7260	123	59.0	0.05s	**0.61**
11	FSM	EDSM	51937	11242	4.6	265.41s	0.55
12	DFA	EDSM	1852	54	34.3	0.01s	**0.68**
12	FSM	Baseline	7549	1270	5.9	2.99s	0.55
13	DFA	EDSM	6613	116	57.0	0.05s	**0.54**
13	FSM	Baseline	66799	12615	5.3	533.00s	0.53
14	DFA	EDSM	5729	108	53.0	0.03s	**0.64**
14	FSM	FA	51503	12706	4.1	357.78s	0.50
15	DFA	FA	2538	78	32.5	0.01s	0.46
15	FSM	States merged	11702	2833	4.1	18.58s	**0.57**
16	DFA	Baseline	1694	43	39.4	0.00s	**0.65**
16	FSM	States merged	9263	1346	6.9	3.65s	0.54
17	DFA	Baseline	3621	78	46.4	0.02s	**0.52**
17	FSM	States merged	21510	4823	4.5	60.33s	0.51
18	DFA	EDSM	2365	58	40.8	0.01s	**0.57**
18	FSM	FA	9906	2014	4.9	7.07s	0.49
19	DFA	EDSM	3117	72	43.3	0.01s	**0.62**
19	FSM	EDSM	15761	3537	4.5	25.60s	0.51
20	DFA	Baseline	1474	43	34.3	0.00s	0.55
20	FSM	States merged	5810	975	6.0	1.54s	**0.58**
21	DFA	EDSM	23733	87	272.8	0.10s	**0.91**
21	FSM	Baseline	140863	15245	9.2	1359.42s	0.57
22	DFA	Baseline	20757	68	305.3	0.08s	**0.94**
22	FSM	EDSM	153546	22679	6.8	2404.96s	0.77
23	DFA	States merged	38132	125	305.1	0.27s	**0.82**
23	FSM	EDSM	315519	40014	7.9	11500.53s	0.53
24	DFA	EDSM	32178	154	208.9	0.52s	**0.67**
24	FSM	EDSM	527975	65984	8.0	42369.99s	0.50
25	DFA	States merged	12635	105	120.3	0.06s	**0.69**
25	FSM	EDSM	73169	9672	7.6	336.56s	0.55

FileID	Model	States in input	States in output	In/Out ratio	Time	Perplexity
1	DFA	82285	17	4840.3	0.06s	43.25
1	SMF	624980	7193	86.9	791.05s	**39.75**
2	DFA	60128	12	5010.7	0.06s	**407.27**
2	SMF	506884	17736	28.6	12603.16s	459.95
3	DFA	39200	21	1866.7	0.03s	69.54
3	SMF	269739	8712	31.0	289.95s	**67.24**
4	DFA	21952	21	1045.3	0.01s	**84.50**
4	SMF	80196	2374	33.8	15.20s	141.38
5	DFA	7204	9	800.4	0.00s	**45.18**
5	SMF	53134	2863	18.6	28.70s	56.82
6	DFA	143410	18	7967.2	0.09s	**88.95**
6	SMF	1835256	46686	39.3	17173.65s	116.75
7	DFA	12689	12	1057.4	0.01s	**76.44**
7	SMF	55010	3525	15.6	86.62s	152.36
8	DFA	252447	76	3321.7	0.28s	**91.91**
8	SMF	1802439	106733	16.9	141634.75s	97.74
10	DFA	178548	33	5410.5	0.26s	58.17
10	SMF	2385293	1779	1340.8	429.84s	**46.99**
13	DFA	187486	59	3177.7	0.17s	**68.55**
13	SMF	1806483	191022	9.5	538487.72s	73.38
14	DFA	55636	7	7948.0	0.07s	205.19
14	SMF	289276	11873	24.4	1980.31s	**183.11**
15	DFA	158761	33	4810.9	0.20s	**71.04**
15	SMF	1833902	38273	47.9	40495.05s	76.93
16	DFA	304258	52	5851.1	0.56s	**32.35**
16	SMF	2373733	110226	21.5	290490.95s	33.83
17	DFA	151853	36	4218.1	0.24s	**71.47**
17	SMF	1767978	56480	31.3	66897.03s	76.84
19	DFA	237802	53	4486.8	0.23s	**18.83**
19	SMF	2648491	105726	25.1	120510.69s	19.35
20	DFA	211599	12	17633.3	0.48s	387.43
20	SMF	2586725	5938	435.6	16905.12s	**314.64**

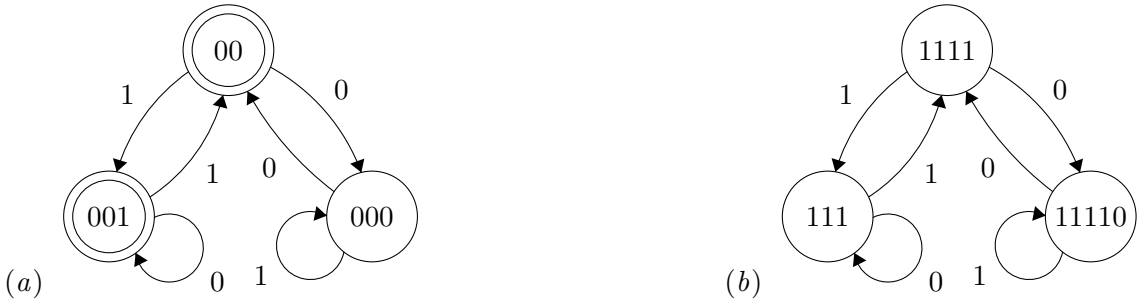

Figure 3: (a) shows the original subtree of the **DFA** for instance 1 of stamina. (b) shows how the simple sink state after prefix 111 would need to be extended in an **SMF**, since prepending 00 to 111 should lead in state 001, while prepending 00 to 1111 should lead to state 00.

Appendix D. Stamina 1 Substructures

Figure 3 shows a simplified example of the substructure blowup that happens in an **SMF**. In the **DFA** 3 (b) can be represented by a single rejecting state with a self-loop, a "sink". The **SMF** however needs to represent this single rejecting state with three states instead, which mirror the structure in (a). This is because 00 could be prepended which would lead to different behaviour depending on the sequence of previously appended 0's and 1's.

To be fully accurate the self-loops centred on $001\backslash 111$ and $000\backslash 11110$ should also be extended into two states. This comes from the observation that $0 \cdot 001 \notin L_1$ while $0 \cdot 0010 \in L_1$, as well as $001 \cdot 000 \cdot 0 \in L_1$ and $001 \cdot 0001 \cdot 0 \notin L_1$. These extensions were omitted to retain readability.