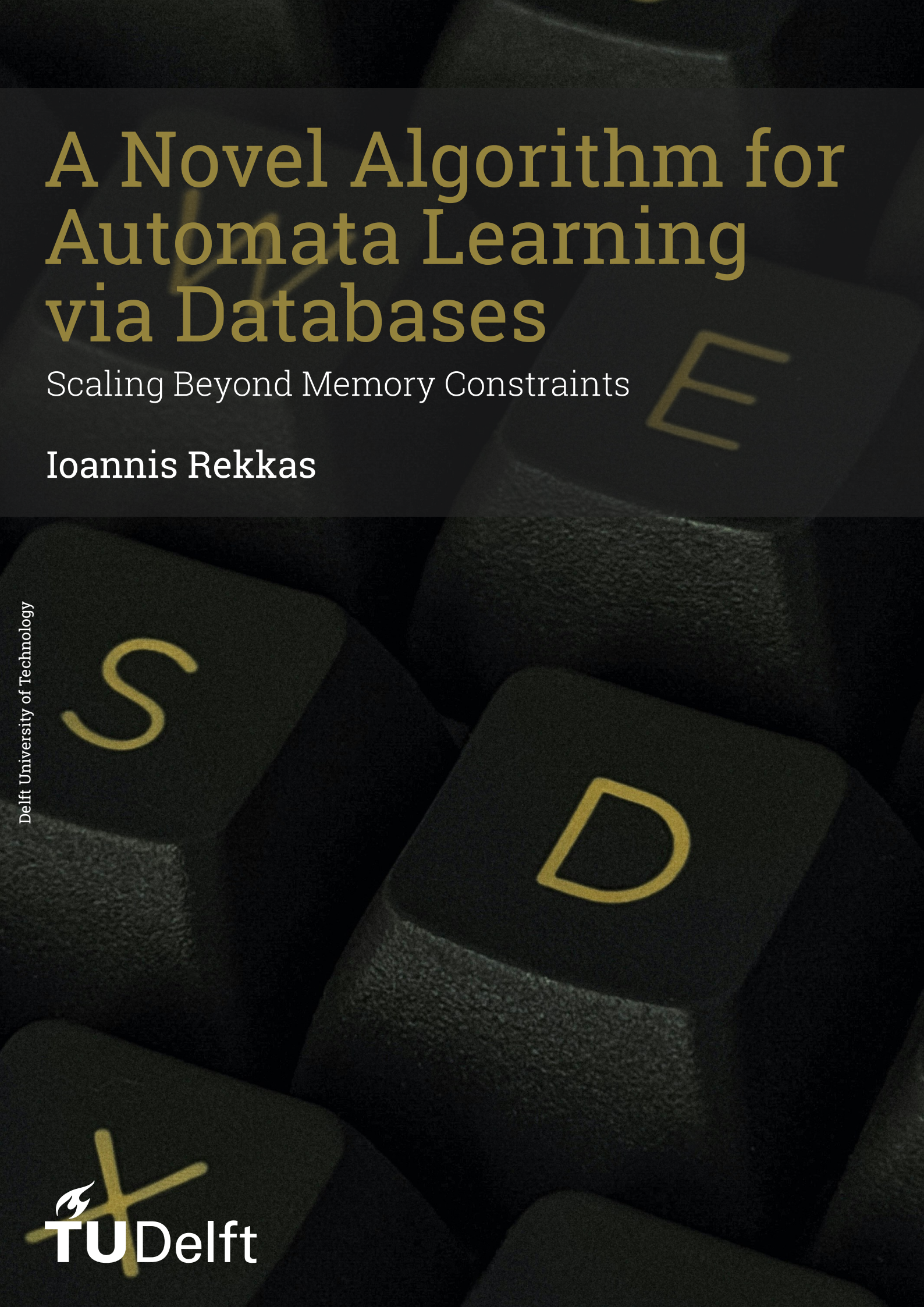
A Novel Algorithm for Automata Learning via Databases

Scaling Beyond Memory Constraints

Ioannis Rekkas



Delft University of Technology

TU Delft

A Novel Algorithm for Automata Learning via Databases

Scaling Beyond Memory Constraints

by

Ioannis Rekkas

Thesis Supervisor: Sicco Verwer
Daily Co-Supervisor: Simon Dieck
Faculty: Electrical Engineering, Mathematics and Computer Science

Cover: *Close-up of black keyboard keys with gold letters*, photo by Qui Nguyen on Unsplash
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Acknowledgments

I would like to thank Professor Sicco Verwer and PhD candidate Simon Dieck, my supervisors. Their expertise in automata learning was highly valuable to my work and helped me gain an in-depth understanding of the field. At the same time, their detailed feedback and high standards significantly elevated the quality of my scientific writing. Their mentorship and patience were instrumental in bringing this project to fruition.

Giannos Rekkas
Delft, July 2026

Summary

Automata learning is a powerful technique for obtaining system models from observed behavior and is often used in software testing, verification, and reverse engineering. However, most algorithms used today either require interaction with that system or suffer from poor scaling due to memory limitations. A relatively new approach in automata learning algorithms has been to store the dataset in a database and interact with it through custom database queries. This thesis uses this approach to develop a novel algorithm based on regular expression queries and custom data structures. Upon evaluation on generated datasets of binary sequences and the Abbadingo benchmarks, this approach was found to scale linearly to much larger dataset sizes than EDSM and to use orders of magnitude fewer queries than L*. It is also accompanied by a formal proof of correctness. Ultimately, this work provides a highly scalable algorithm and a self-contained, comprehensive theoretical framework.

Contents

1

Introduction

The rapid growth of software services, distributed systems, embedded devices, and artificial intelligence has created multitudes of black-box components whose inner workings are difficult or impossible to understand and explain comprehensively. Developers and system maintainers regularly collect large volumes of log data and execution traces, typically stored in large-scale databases. That data contains useful information about the system's operation, but manually forming an accurate model is next to impossible and error-prone.

Automata learning (also referred to as regular inference) gives a structured solution. It automatically infers finite-state models, usually deterministic finite automata (DFAs), from the observed behavior of the system. These models are quite useful in reverse engineering [10, 3], formal verification, anomaly detection [6], testing [1], and debugging. However, current methods exhibit severe scalability limitations when used in real-world big-data scenarios.

Active learning algorithms, such as L* [2], can obtain an exact machine quickly but need interaction with the system under learning (SUL). In most production environments, this would be either impractical or plainly impossible. Passive learning methods, such as RPNI [9], use static datasets but require the whole dataset to be loaded in memory, something that obstructs scaling and makes it impossible for them to work with big data. In contrast, most software systems in use are large and can produce millions of logs per second. This makes the use of these algorithms impossible in such cases, as such amounts of data cannot fit in the largest of modern RAMs.

This thesis overcomes such challenges by using a *database-assisted learning* approach. The main idea is to store almost all the dataset traces in a database and interact with it via queries as needed for the learning process. Moreover, the specific algorithm implemented here uses regular expressions for that task, retrieving suffixes that indicate new states and expanding the exploration frontier without fully loading the dataset into memory.

This method has several advantages:

- **Scalability**: It can work with datasets orders of magnitude larger than what passive learning algorithms can handle.
- **Practicality**: It works directly with existing log infrastructure (SQL databases).
- **Connection to Theory**: It builds on well-established results in theoretical computer science, like the Myhill-Nerode theorem.
- **Hybrid Approach**: It combines ideas from passive learning and active learning into an efficient algorithm.

Modern relational databases are equipped with widely expressive query languages. They can be used to program an arbitrary oracle. An interesting function provided in most of them, `regexp_matches`, is going to be of interest in this thesis, as it can be used, as we will see, to detect *distinguishing suffixes* which can differentiate candidate states.

L* makes individual membership queries for all prefixes and suffixes. The same information can be retrieved with a single SQL query using a regular expression containing the prefixes and suffixes.

1.1. Contributions

The main contribution of this thesis is a new database-assisted automata learning algorithm that uses regular expression queries to split equivalence classes. I present the theoretical foundation, detailed algorithm design, and empirical evaluation on large datasets, and correctness arguments. Results indicate scalability where passive learning methods fail, model accuracy comparable to active methods, and orders of magnitude higher query efficiency.

By enabling automata learning at true big-data scales, this work opens new possibilities for extracting models from production logs, legacy systems, industrial control environments, and a multitude of domains where accurate models are difficult to obtain either because of large datasets or limited interaction capabilities with the system.

1.2. Societal and Temporal Impact

The ability to obtain accurate models from massive volumes of execution traces and logs can have a large impact on modern society. Nowadays, software systems keep growing in size and complexity. At the same time, they are becoming critical infrastructure as they power more and more core parts of social and economic activity, such as financial systems, healthcare, power grids, supply chains, AI, and, in the near future, autonomous vehicles and perhaps smart cities. As such, the need for reliable and interpretable models of system behavior becomes imperative. The database-assisted approach can fulfill that need, especially in the age of big data. Whereas traditional methods struggle to scale, this method can use real-world log data directly and does not require loading the entire dataset or interacting with the system.

Moreover, in the current era of rapid AI development, automata learning can function as a complementary approach. Neural networks perform well on predictive tasks in statistical datasets, but they often lack interpretability or formal guarantees. Automata give interpretable and verifiable models, and perhaps could even be used on AI black-box models to derive explanations.

In the coming years, as edge computing, IoT, and AI-powered systems continue to expand, scalable model learning can improve system trustworthiness and explainability. This research thus promotes more robust digital infrastructure and further innovation in automata learning.

1.3. Research Questions

This thesis, while developing this database-assisted algorithm, answers some research questions concerning it.

The research questions guiding this thesis are:

- How does such an algorithm compare with typical passive learning methods in terms of runtime and scalability?
- How does it compare with active learning methods in terms of query efficiency?
- How can the algorithm be expanded with the aim of discovering more states autonomously, without specific human input or guidance?
- How does the expanded algorithm compare in the same ways?
- Can we give a correctness proof for the algorithm?

1.4. Thesis Outline

Chapter 2 covers the required theoretical background in automata theory and automata learning. Chapter 3 develops a simple form of the algorithm and demonstrates the desired properties. Chapter 4 expands that version with new data structures and methods to autonomously explore traces and discover previously missed states of the automata. Chapter 5 then describes the evaluation methodology and synthetic datasets. Chapter 6 presents the experimental results. Chapter 7 analyzes the correctness

of the algorithm and provides a correctness proof. Finally, chapter 8 concludes by summarizing the answers to the research questions, covering the limitations, and outlining possible future work.

2

Background

This chapter reviews the theoretical concepts on which this thesis relies. It begins with a formal description of DFAs, a review of active and passive learning algorithms, the contributions of database-assisted learning, the Myhill-Nerode theorem accompanied by a proof sketch, and brief details on regular expressions in databases.

2.1. Notation and Definitions

First, we recall standard notation and definitions necessary for automata theory.

Let Σ be a finite alphabet, that is, a finite and nonempty set of symbols. A *string* over Σ is a finite sequence of symbols of Σ. The unique string of length zero is called the *empty string*, and it is symbolized with ε.

Concatenation is an operation of strings, and it is usually denoted either by juxtaposition or with a dot. For instance, $x = a_1a_2 \dots a_n, y = b_1b_2 \dots b_m$, their concatenation is $x \cdot y = a_1a_2 \dots a_nb_1b_2 \dots b_m$ (where $a_i, b_i \in \Sigma$).

Given a set S of strings over Σ, the (right) extension of S is defined as $S\Sigma = \{s \cdot a \mid s \in S, a \in \Sigma\}$. This notation is sometimes used in automata learning algorithms (algorithm 1).

Using the above notation, we can also write $\Sigma^k = \Sigma^{k-1}\Sigma$, $\Sigma^0 = \{\varepsilon\}$, and

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$$

The set of all possible strings over Σ is symbolized as Σ^*. A *language* in computer science is defined as a set of strings of an alphabet and symbolized as $L \subseteq \Sigma^*$. The languages accepted by some deterministic finite automaton are called *regular*.

2.2. Deterministic Finite Automata

A DFA (Deterministic Finite Automaton) is a state machine with finite states and a single transition per state and input symbol to another state. An example is the state machine of figure 2.1. It accepts multiples of 5 using the known rule that they must end with 0 or 5.

When depicting DFAs, typically, the *initial state* is marked with an incoming arrow from nowhere, and the *accepting* (also called *final*) states are marked with a double circle. In the DFA of figure 2.1, either state can function as the initial state, so I did not mark one.

The formal definition of a DFA is as follows. A *deterministic finite automaton* (DFA) is a 5-tuple [13, 5]

$$M = (Q, \Sigma, \delta, q_0, F),$$

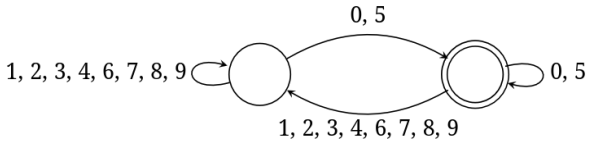


Figure 2.1: A DFA accepting multiples of 5.

where

- Q is a finite set of states,
- Σ is a finite set called the input alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of accepting (final) states.

A string $w = a_1 a_2 \dots a_n \in \Sigma^*$ is accepted by the DFA M if there exists a sequence of states $r_0, r_1, \dots, r_n \in Q$ such that $r_0 = q_0$, $r_{i+1} = \delta(r_i, a_i)$ for all $0 \leq i < n$, and $r_n \in F$. The *language recognized* by M, denoted $L(M)$, is the set of all strings that M accepts.

2.3. Automata Learning

Automata learning, sometimes additionally referred to as regular inference or model learning, is a field of computational learning theory and formal methods that encompasses inferring finite-state models (like the DFA already described) from observations of a system. The aim is to automatically construct an explainable and accurate model of a system. This is quite useful in cases where the system is a black box, and it has found applications in software testing, reverse engineering, and communication protocols [3].

Automata learning algorithms are usually classified in two main categories based on the way they interact with the system: *active learning* and *passive learning*. Active learning uses an oracle that answers queries, while passive learning uses datasets of labeled examples. Several libraries that implement those algorithms have arisen, such as LearnLib [11], AALpy [8], and FlexFringe [14].

2.4. Active Learning

Active learning covers cases where the learner interacts with the target system (sometimes referred to as the *System Under Learning*, or SUL, for short). The first algorithm for active automata learning was introduced in 1987 by Angluin [2]. In it, the oracle answers two queries:

- *Membership queries*: Given a string $w \in \Sigma^*$, the oracle answers whether $w \in L$, the target language.
- *Equivalence queries*: Given a hypothesis automaton H, the oracle answers whether $L(H) = L$, and if it is not the case, it gives a counterexample $w \in L \setminus L(H)$.

The algorithm using these oracles is called L*, and it runs in polynomial time of the DFA size and of the length of the longest counterexample.

The algorithm operates by using an *observation table* consisting of prefixes and suffixes. It fills that table with the membership queries, uses it to build a hypothesis that is consistent with the observations, and asks an equivalence query, repeating this process in case the hypothesis wasn't correct.

The observation table is expressed as a triple (S, E, T) where:

- $S \subseteq \Sigma^*$ is a set of prefix strings (representatives of states),
- $E \subseteq \Sigma^*$ is a set of suffix strings (distinguishing the prefixes, see section 2.6),
- $T : (S \cup S\Sigma) \times E \to \{0, 1\}$ is the membership table, where $T(s, e) = 1 \iff s \cdot e \in L$

The row of a string s corresponds to the function $\mathrm{row}(s) = (T(s,e))_{e \in E}$. We may write $\mathrm{row}(s_1) = \mathrm{row}(s_2)$ meaning $\forall e \in E, T(s_1, e) = T(s_2, e)$

The table is called *consistent* if $\mathrm{row}(s_1) = \mathrm{row}(s_2) \implies \forall e' \in E, \mathrm{row}(s_1 \cdot e') = \mathrm{row}(s_2 \cdot e')$. It's called *closed* if every row of a string $\in S\Sigma$ also exists in S.

Algorithm 1 Angluin's L* Algorithm

Require: Alphabet Σ, membership and equivalence oracles
1: Initialization $S \leftarrow \{\varepsilon\}$, $E \leftarrow \{\varepsilon\}$
2: Fill the observation table T for $(S \cup S\Sigma) \times E$ using membership queries
3: **repeat**
4: **while** the observation table (S, E, T) is not *closed* or not *consistent* **do**
5: **if** (S, E, T) is not consistent **then**
6: Find $s_1, s_2 \in S$, $a \in \Sigma$, $e \in E$ such that $\mathrm{row}(s_1) = \mathrm{row}(s_2)$ but $T(s_1 a, e) \neq T(s_2 a, e)$
7: $E \leftarrow E \cup \{ae\}$
8: Use membership queries on $\forall s \in S \cup S\Sigma, s \cdot ae$ and record the answers in T.
9: **end if**
10: **if** (S, E, T) is not closed **then**
11: Find $s \in S$, $a \in \Sigma$ such that $\mathrm{row}(sa)$ is different from $\mathrm{row}(s')$ for all $s' \in S$
12: $S \leftarrow S \cup \{sa\}$
13: Use membership queries on $\forall e \in E, sa \cdot e$ and $\forall a' \in \Sigma, saa' \cdot e$ and record the answers in T.
14: **end if**
15: **end while**
16: Construct the hypothesis DFA H from the closed and consistent table (S, E, T)
17: Ask an equivalence query with H
18: **if** the teacher replies "yes" **then**
19: **return** H
20: **else**
21: Receive a counterexample $w \in L \setminus L(H)$
22: $S \leftarrow S \cup \mathrm{Pref}(w)$ ▷ all prefixes of w (see following text)
23: Extend T to $(S \cup S\Sigma) \times E$ using membership queries
24: **end if**
25: **until** the equivalence query is answered positively

An interesting detail regarding the practical and theoretical efficiency of L* is indicated in line 22 of algorithm 1 and concerns the handling of counterexamples. When the hypothesized DFA is not correct, the oracle returns a counterexample $w \in L \setminus L(H)$.

In Angluin's original algorithm, all prefixes of the counterexample are added to S. An important optimization was discovered later [12]:

Given a counterexample w, it suffices to identify and add a single suffix of w to E instead of all prefixes to S. To the best of our knowledge, that suffix can be found using binary search in the counterexample, thus requiring only $O(\log |w|)$ membership queries[1].

This method drastically reduces the number of membership queries. In practice, this is the standard implementation in modern automata learning libraries.

2.5. Passive Learning

In contrast to active learning, passive learning infers a model from a finite set of labeled examples without interacting with the target system. These labeled examples could, for instance, come in the form of software logs.

[1] I was informed that some soon-to-be-published work might refute that claim and show that the optimization still yields smaller improvements nevertheless.

One of the first passive DFA learning algorithms is called **RPNI** (Regular Positive and Negative Inference) [9]. RPNI works by constructing a prefix tree from the positive samples and then iteratively merging states while remaining consistent with the negative samples. It uses the red-blue framework where red nodes are finialized while blue ones are being merged.

Algorithm 2 RPNI

Require: Positive sample $S^+ \subseteq \Sigma^*$, negative sample $S^- \subseteq \Sigma^*$
Ensure: DFA consistent with S^+ and S^-

1: Build the *Prefix Tree Acceptor* (PTA) from S^+ ▷ Tree-shaped DFA accepting exactly S^+
2: *red* $\leftarrow \{\varepsilon\}$ ▷ Set of fixed (red) states
3: *blue* $\leftarrow$ {one-symbol extensions of strings in red not already in red}
4: **while** *blue* $\neq \varnothing$ **do**
5: Choose the next blue state b according to a fixed order
6: *merged* $\leftarrow$ false
7: **for each** red state $r \in$ *red* (in a suitable order) **do**
8: **if** merging b into r is consistent with S^- **then**
9: Perform the merge
10: For any nondeterminism (same symbol transitions to different states after merge) keep merging these states recursively until it's gone.
11: *merged* $\leftarrow$ true
12: **break**
13: **end if**
14: **end for**
15: **if not** *merged* **then**
16: Promote b to red ▷ It becomes a new distinct state
17: **end if**
18: Update *blue* to include new one-symbol extensions of red states not already present
19: **end while**
20: **return** the resulting DFA

As we can see in lines 5 and 7 of algorithm 2, RPNI examines the red and blue nodes in arbitrary order. One improvement of this is to score each pair of red and blue nodes by a heuristic of how likely they are to merge and examine them in that order. This exact idea is implemented by the EDSM algorithm, which stands for Evidence-Driven State Merging [7].

EDSM is known for often achieving better generalizations on real-world datasets, and it came first in the Abbadingo one DFA learning challenge [7] (also see section 2.8).

Passive learning is especially useful in situations where only logs or historical data are available, or when active interaction with the system is either impossible or very expensive. However, this setup has been proven to be NP-hard [4].

The comparison of the two paradigms, active and passive learning, is usually in practical terms. Active learning typically requires fewer total examples but requires interaction with an oracle, while passive learning can yield good approximations if the dataset is sufficiently large or high-quality.

2.6. Database-Assisted Automata Learning

A recent line of research has proposed a solution to scalability issues arising from large datasets in passive learning algorithms. Such datasets are commonly found in real-world software systems in the form of logs, which are usually stored in databases. Most traditional passive learning algorithms, including the aforementioned RPNI and EDSM, require that the whole dataset be loaded in RAM for the prefix tree acceptor. This becomes infeasible for large datasets that cannot fit into RAM. Some solutions, including sketching or hashing algorithms, have been proposed for this issue. However, here we examine the database-assisted approach.

A proposed solution is **DAALder** [15] (from Database-Assisted Automata Learning), a hybrid approach combining active and passive learning. Instead of loading the whole dataset in memory or using in-

teractive queries, DAALder stores the traces in a database and uses database queries in iterations, expanding an observation tree.

DAALder innovates this space by inventing a *prefix query* which retrieves traces from the database that have a common prefix. It works as follows: given a prefix trace $t \in \Sigma^*$, a maximum length $n \in \mathbb{N}$, and a number $k \in \mathbb{N}$, $\mathrm{PrefixQuery}(t, n, k)$ returns up to k traces (with their outputs) from the database that start with the prefix t and have a length of at most n. Furthermore, this query was implemented in SQL [15].

DAALder uses the red-blue framework for state merging. It keeps a prefix tree acceptor, and merges states using heuristics similar to those in EDSM. If these heuristics score low enough that the merges are considered uncertain, it uses prefix queries to obtain more traces and evidence for or against merges. Then it uses equivalence oracles to validate the hypothesized automaton.

The prefix queries have several effects for DAALder:

- It enables the efficient retrieval of multiple traces that extend a given prefix without using the entire dataset.
- In the aforementioned state-merging framework that DAALder also uses (see the section on passive learning), they can be used when merges are uncertain (multiple red states having similar scores for a blue state) to obtain more information, which alters the scores and resolves ambiguity.
- The maximum length n and number of results k function as limits to the amount of prefixes being processed, allowing the queries to be computationally fast while still providing useful information fro state-merging.

Since the queries of the algorithm in this thesis are different, these effects are not general to database-assisted learning but rather specific to DAALder.

This technique offers two main advantages:

- **Scalability**: Only a small part of the dataset is loaded in RAM at a given moment, with the database handling storage and retrieval.
- **Correctness**: If a *characteristic sample* is in the database, DAALder converges to the target automaton.

Experimental results on large synthesized datasets (up to tens of millions of traces) show that DAALder achieves similar levels of accuracy to EDSM but that EDSM uses exponential memory with respect to the dataset size, while DAALder uses constant memory (a few gigabytes), making it quite fitting for use with real-world log data. Expectedly, however, DAALder is slower than EDSM when the latter doesn't require the usage of swap space.

Database-assisted learning is said to lie between passive learning on static datasets and active learning on interactive systems, thus enabling practical model learning in big data situations common in software engineering.

Another important idea from DAALder is that the database approach allows one to design custom queries for the learning algorithms (corresponding to database queries) and to use them instead of the traditional membership/equivalence queries of the L* algorithm.

2.7. Myhill-Nerode Theorem

The Myhill-Nerode theorem uses algebraic properties of regular languages and proves the size of the minimal target DFA for each regular language. It is a fundamental result in automata theory, and it connects the structure of a language to its minimal DFA [5, 13].

To express the theorem, an equivalence relation is defined as follows. Let $L \subseteq \Sigma^*$ be a language over the alphabet Σ. The equivalence relation $\sim_L$ is defined on Σ^* as

$$x \sim_L y \iff \forall z \in \Sigma^*, (xz \in L \iff yz \in L).$$

The $\sim_L$ relation is also referred to as the right equivalence induced by L. The number of equivalence classes in $\sim_L$ is referred to as the *index* of this relation.

Then the theorem can be stated as follows. A language $L \subseteq \Sigma^*$ is regular if and only if the index of $\sim_L$ is finite. Moreover, if L is regular, then the number of states in the minimal DFA accepting L is exactly equal to the index of $\sim_L$.

Proof Sketch
($\Rightarrow$) If L is accepted by a DFA with n states, then strings that end up in the same state are equivalent under $\sim_L$ (because any extension leads them to the same states again), meaning there are at most n equivalence classes.

($\Leftarrow$) If $\sim_L$ has finite index n, set each equivalence class to a state of the DFA. The transition function is just right-concatenation with a symbol (because we can set $z = az'$ and since x, y have the same outcome for every z, xa, ya have the same outcome $\forall z'$, meaning all one-symbol extensions belong to the same class). The initial state is the class of ε, and accepting states are the classes that contain a string of L. This is the unique (up to isomorphism) and minimal DFA or L.

The Myhill-Nerode theorem has some interesting implications for automata learning:

- It gives a lower bound on the size of any DFA recognizing a language.
- The equivalence classes correspond to the “states” discovered by automata learning algorithms.
- It is often used in correctness proofs for the algorithms.

For automata learning purposes, the theorem can be summed up in the following way. For strings x, y, if there is no suffix z such that xz accepts, and yz rejects (or vice versa), then x and y belong to the same state in the DFA.

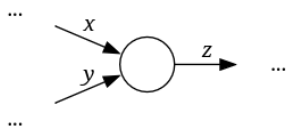


Figure 2.2: Demonstration of Myhill-Nerode, if x and y lead to the same state, then with any suffix z they should also lead to the same state

This theorem is the backbone of every automata learning algorithm, and these algorithms try to use queries that would demonstrate a violation of the theorem to split states or to prove they can’t join states.

2.8. The Abbadingo Challenge

An important initiative in automata learning was the *Abbadingo Challenge* (also known as Abbadingo One). It was organized in the late 1990s with the aim of promoting research in automata learning, and particularly passive learning of DFAs from labeled examples [7]. Named after a character in a classic sci-fi novel, the challenge consisted of a benchmark of several learning problems of various dataset sizes, DFA depths, and state numbers, in a binary alphabet ($\{0, 1\}$). The participants were given datasets with positive and negative examples generated by unknown DFAs and had to infer a correct automaton that generalized to unseen data.

The Abbadingo Challenge contributed greatly to the advancement of passive learning algorithms. It enabled the examination of the strengths and limitations of early state-merging algorithms and led to significant improvements, such as the aforementioned Evidence-Driven State Merging algorithm. Many modern passive learning algorithms, such as extensions of RPNI, trace their origins to insights from this competition.

2.9. Regular Expression Queries in Databases

As discussed, in database-assisted automata learning, a crucial operation is the retrieval of traces from large datasets in a database. Since we want to select useful traces for learning, we might want to retrieve ones that match certain patterns. This can be easily done with *regular expression (regex) queries*.

A regular expression over an alphabet Σ defines a regular language and can be represented by a DFA. Modern database management systems support regular expression (regex) matching. This is often implemented by transforming the expression into an automaton and matching the stored traces. Many engines provide multiple ways of performing such operations (i.e., PostgreSQL has `SIMILAR TO` and `~`).

Such an approach combines theoretical automata learning techniques with practical big-data infrastructure, allowing learning algorithms to operate on terabyte-scale trace logs without loading the whole dataset in memory.

2.10. Basic Regular Expression Operators

A little bit of regular expressions will be used in some parts of the algorithm, so the basics required for understanding them are presented here.

Literals
Every symbol $a \in \Sigma$ is a regular expression denoting the language that only contains itself $\{a\}$.

Concatenation
If r_1 and r_2 are regular expressions, then $r_1 r_2$ denotes the language $\{xy \mid x \in L(r_1),\ y \in L(r_2)\}$.

Union
If r_1 and r_2 are regular expressions, then $r_1|r_2$ denotes $L(r_1) \cup L(r_2)$.

Kleene star
If r is a regular expression, then r^* denotes the language $\bigcup_{k=0}^{\infty} L(r)^k$, i.e., finite concatenations of strings from $L(r)$, including the empty string ε (L^k is defined as $L^{k+1} = \{xy \mid x \in L^k, y \in L\}$ and $L^0 = \{\varepsilon\}$).

Anchors

- The caret `^` asserts the beginning of a line.
- The dollar sign `$` asserts the end of a line.

Together, `^r$` matches strings that exactly match the pattern r (no extra characters before or after).

Parentheses are used to group expressions, and depending on the implementation, they also mark *capture groups*. Some regex functions can return only the capture group rather than the entire matching line.

Example The regex `^(a|b)(c|d)*$` matches lines that start with either `a` or `b` and are followed by any number of `c` or `d`: `a, b, ac, bc, ad, bd, acc, acd, adc, add, bcc, bcd, bdc, ....` The first capture group is the first set of parentheses and in any case captures `a` or `b`. The second capture group in the second parenthesis may capture ε, `c, d, cc, cd, dc, dd, ....`

Depending on the implementation, if we omitted `$` then lines like `abcbcb555` would also be matched. Similarly, if we omitted `^`, we might match `555bccc`.

2.11. PostgreSQL

PostgreSQL was selected as the DBMS (database management system) for the algorithm implementations because of the richness of its query languages. A relevant such expression for the algorithm is demonstrated in this section.

2.11.1. Regular Expressions

PostgreSQL provides `regexp_matches(`*`text`*`, `*`pattern`*`)`, a function which can be used to get the substrings matched in the capture groups of the pattern. In the algorithm, this function is used to both match a prefix and to get back the suffix in the same query for further processing. This is demonstrated in listing 2.1.

Listing 2.1: Prefix and suffix decomposition.

```
SELECT (m)[1] AS prefix,
       (m)[2] AS suffix,
       accepting
FROM   traces,
       regexp_matches(trace, '^(100|101)(0|1)*$') AS m;
```

Here, the two prefixes, `100` and `101`, are searched for and matched only when the suffix is a sequence of `0`s and `1`s. At the same time, the two capture groups marked by the parentheses in the regex mark the prefix and suffix, which are recovered by indexing the result of the function in `(m)[1]` and `(m)[2]`. Note that we assumed we have a table called `traces` with two columns: `trace`, which contains the actual trace as a string, and `accepting`, which indicates whether the string is accepted or rejected by the target DFA.

2.12. Summary

In this chapter, some fundamental theoretical aspects and ideas for automata learning were presented. It covered the L* that algorithm interacts with the system to derive a DFA. The Myhill-Nerode theorem, which underlies automata learning algorithms, enables the construction of minimal DFAs for each regular language. Passive learning algorithms all need to keep the entire dataset in memory, which becomes a bottleneck and, in large enough datasets, causes crashes. Database-assisted learning, on the contrary, avoids this by utilizing databases and keeping the data in the larger capacity of the hard disk. The Abbadingo challenge enabled deeper passive learning research. Modern functions offered by database management systems can serve as powerful queries for learning algorithms, with regex matching being particularly useful for both checking acceptance of traces and splitting them into prefixes and suffixes at the same time.

3

Simple Split Loop

This chapter demonstrates the development of a simple form of my novel database-assisted learning algorithm. Then it shows that this version already has the desired properties (scalability, query efficiency).

3.1. The Idea

The main idea of the basic split loop structure is to start with a set of prefixes in an equivalence class, run a database query to detect distinguishing suffixes using some regular expression of the prefixes of that class, and split into further classes if one is found. If it is not found, since we do not yet add new prefixes, that class is finalized, and we don't need to query it anymore. Then repeat this for every class.

3.2. Simple Utilities

First, we develop some simple utilities that help express the main part of the algorithm with greater ease, abstracting some details that are dealt with in these utilities. They encapsulate the formation of the regular expression of the query in algorithm 3 and the splitting of a class with the database query in algorithm 4.

The way of obtaining a regular expression for all prefixes that end up in an equivalence class is creating an OR expression with all the members of the class, followed by a pump of all the possible extensions already in that class (because if we see a and abb in the same class, we can infer that bb leads back to the same class, so anything in that class followed by bb is also in the same class). In example, for the class $\{a, b, aa, abb\}$, the regex would be `/^(a|b|aa|abb)(a|bb)*$/`. Notice that due to a and aa being in the class, a is an extension, and for the same reasons, bb is an extension. The process is expressed in algorithm 3.

A disadvantage of this query is that it entirely relies on having a good and representative sample of prefixes in the class.

Advantages of this Regex Query

A regex query drastically reduces the number of queries one makes since, for instance, membership queries can be batched in a regex, and also the total time, since an intelligent implementation could perform parts of it in parallel. This specific query pumps the suffixes because this limits the matched traces and so reduces the subsequent workload of the database.

The database query is expressed as QueryDB(r, label), where r is the regular expression we are querying for, and label is the desired label (accepted or rejected). After gathering all matches of $r =$ `^(prefixes)(suffixes)$`, it selects the shortest of the possible suffixes and returns the prefixes of the class (constrained in the r) that were followed by that suffix and whose label matched the requested one (the opposite of the class so that it returns counterexamples). The SQL implementation of this query is presented in Appendix A.

Algorithm 3 Obtaining a Regular Expression from an Equivalence Class

function $\mathrm{GetRegex}(C)$
  $R \leftarrow \emptyset$ ▷ Base strings in the class
  $P \leftarrow \emptyset$ ▷ Pumpable (periodic) suffixes
  for each $x \in C$ **do**
    $R \leftarrow R \cup \{x\}$
    for each proper split of x in to a prefix-suffix pair (p, s) **do**
      if $p \in C$ **then**
        $P \leftarrow P \cup \{s\}$
      end if
    end for
  end for
  return $\hat{}\left(|_{x \in R}\ x\right)\left(|_{p \in P}\ p\right)^{*}\ \$$ ▷ $\hat{}, |, ^{*}, \$$ are the standard regex operators.
end function

The procedure SplitClass(C) (algorithm 4) splits the class C if a Myhill-Nerode counterexample is found through the query. If it did not find a counterexample, then this is signaled to its caller with the return of $\emptyset$ (whereas normally it returns the two new classes).

Algorithm 4 Splitting an Equivalence Class Using Database Queries

function $\mathrm{SplitClass}(C)$
  $r \leftarrow \mathrm{GetRegex}(C)$
  $C_{\text{opp}} \leftarrow \mathrm{QueryDB}(r, \text{label} \neq C.\,\text{label})$ ▷ Strings with opposite label
  $C_{\text{same}} \leftarrow C \setminus C_{\text{opp}}$
  if $C_{\text{opp}} \neq \emptyset$ **and** $C_{\text{same}} \neq \emptyset$ **then**
    return $\{C_{\text{opp}}, C_{\text{same}}\}$
  else
    return $\emptyset$
  end if
end function

Example Consider the multiples of 5, initially all in one rejecting class: $C = \{0, 1, 2, \ldots, 10\}$. Running split, we obtain $C_{opp} = \{0, 5, 10\}$ (with the ε suffix) and as such $C_{same} = \{1, 2, 3, 4, 6, 7, 8, 9\}$, SplitClass returns two new classes, the first accepting, the second rejecting: $\{\{0, 5, 10\}, \{1, 2, 3, 4, 6, 7, 8, 9\}\}$.

If we try to run SplitClass on $\{0, 5, 10\}$, we do not find a distinguishing suffix (all of them are in the same class because they are multiples of 5). $C_{opp} = \emptyset$ this time, and as such, SplitClass tells us this by returning $\emptyset$.

3.3. Basic Split Loop

Finally, with these tools, we can define the main loop of the algorithm (algorithm 5). It starts with an initial class of small prefixes from the dataset (length of up to 2-5 characters). Then it repeatedly splits this class and the new ones until the classes cannot be split again. Finally, the classes are converted into a DFA with a method explained in the subsequent section.

Example Let's consider the multiples of 5 again, starting with the same class as previously.

First iteration: $\mathcal{P} = \{\{0, 1, 2, \ldots, 10\}\}, S = \{\{0, 5, 10\}, \{1, 2, 3, 4, 6, 7, 8, 9\}\}, \mathcal{F} = \emptyset$.

Second iteration: $\mathcal{P} = \{\{0, 5, 10\}, \{1, 2, 3, 4, 6, 7, 8, 9\}\}, S = \emptyset$ (tried to split $\{0, 5, 10\}$), $\mathcal{F} = \{\{0, 5, 10\}\}$.

Third iteration: $\mathcal{P} = \{\{1, 2, 3, 4, 6, 7, 8, 9\}\}, S = \emptyset$ (tried to split it but it's also one class since the minimal DFA has two states), $\mathcal{F} = \{\{0, 5, 10\}, \{1, 2, 3, 4, 6, 7, 8, 9\}\}$.

Fourth iteration: $\mathcal{P} = \emptyset$, end while.

Algorithm 5 Basic Main Loop of the Database-Assisted Learning Algorithm

Require: Database connection, C_0: initial equivalence class of short prefixes
Ensure: DFA constructed from final equivalence classes

$\mathcal{P} \leftarrow \{C_0\}$ ▷ Classes still to be processed
$\mathcal{F} \leftarrow \emptyset$ ▷ Classes that could not be split further
while $\mathcal{P} \neq \emptyset$ **do**
  $C \leftarrow \mathcal{P}.\,\mathrm{pop}()$
  $S \leftarrow \mathrm{SplitClass}(C)$ ▷ Either $\emptyset$ or $\{C_1, C_2\}$
  if $S \neq \emptyset$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup S$
  else
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{C\}$
  end if
end while
return $\mathrm{BuildDFA}(\mathcal{F})$

So this loop correctly identifies the equivalence classes for multiples of 5, and they are passed to the DFA constructor. Were it a more complicated automaton, the loop would have kept splitting the second class.

3.4. Converting the Sets of Equivalence Classes to a DFA

The conversion to a DFA follows exactly the method described in the Myhill-Nerode theorem (section 2.7). It consists of finding a small member of each class (a representative) and determining which class its 1-letter extensions belong to. If the representative is small, its extensions are more likely to be in our sets. Then we can connect the state of that class to the state of the other class via a transition labeled with that letter in the DFA. The final states are easy to find through the classes computed label, and the initial state is the one that contains ε (algorithm 6).

Algorithm 6 Converting a Set of Equivalence Classes into a DFA.

function $\mathrm{BuildDFA}(\mathcal{F})$
  for each $C_i \in \mathcal{F}$ **do** ▷ Iterate through all equivalence classes
    Create a state $q_i \in Q$
    if $\varepsilon \in C_i$ **then** ▷ Identify the initial state
      $q_0 \leftarrow q_i$
    end if
    if $C_i.\,\mathrm{label} = \top$ **then**
      $F \leftarrow F \cup \{q_i\}$
    end if
    Choose a small representative $u \in C_i$
    for each $a \in \Sigma$ **do**
      Find class $C_j \in \mathcal{F}$ such that $u \cdot a \in C_j$
      Create a transition $\delta(q_i, a) \leftarrow q_j$
    end for
  end for
  return $(Q, \Sigma, \delta, q_0, F)$
end function

3.5. Initial Experiments On This Form of the Algorithm

Research Question How does this algorithm compare with passive and active learners?

We can use this simple version of the algorithm as a proof of concept and run some initial experiments to demonstrate the properties we wish to see. Mainly, that it can learn automata in reasonable time frames and that it can scale past the RAM limit of passive learning. Later, we will also compare it to

active learning.

3.5.1. Datasets

In order to evaluate the algorithm, we need appropriate datasets. As already discussed, we want to explore how the algorithm behaves with extremely large datasets. An issue with that is that such datasets are rarely public. For that reason, we will follow the practice of Walinga et al. [15] and generate large datasets.

For testing this version, several datasets of about 1.5 million traces were generated. They were datasets concerning multiples of 3, 5, and 7 in decimal as well as binary. Multiples of a number (equivalence classes modulo n) are a classic theoretical application of DFAs. They are also quite easy to generate in large scales.

3.5.2. Demonstrations of Resulting DFAs

The proof of concept was implemented in Python with PostgreSQL for the database. It is, for the most part, a direct translation of the pseudocode presented. It can also output the Graphviz code for the automaton it computes. Several interesting results are depicted in the figure 3.1. They showcase that the algorithm learns the correct automata for each dataset.

3.5.3. Time

The execution time of this algorithm depends on the structure of the automaton being learned. The bottleneck is the query, and more queries mean drastically slower executions for large traces. While this will be expanded in the next section, it is important to keep it in mind when investigating the execution times of the experiments. The dataset size is in each case 1.5 million traces.

Multiples of	**3**	**5**	**7**
Time (Binary)	83.98s	108.51s	109.89s
Time (Decimal)	13.54s	10.70s	19.26s

Table 3.1: Execution times over 1.5 million trace datasets

To get better results in binary datasets, the set of initial traces used prefixes of length up to 6, whereas in decimal, only up to 2. One can easily see that the decimal DFA of decimal multiples of 5 was the fastest, as it only has 2 states. Moreover, the larger trace size of the binary examples creates more complicated regular expressions, which are slower when checking for a match.

3.6. Analysis

3.6.1. Issues

The slowest part of the proposed approach is the database query. Regular expression matching against millions of traces is a slow process in most modern database management systems. For this reason, minimizing the number of queries can be really beneficial. However, this cannot be done with this exact approach. A different approach, which uses the query as the last resort, will be sketched in the last section of this chapter.

3.6.2. Hotspot Analysis

To analyze how the algorithm behaves in terms of runtime, I instrumented the database query of the proof of concept and recorded the time it took each time it was executed. After running for each binary dataset of 1.5 million traces 100 times, the results indicate the first query is responsible for over 50% of the execution time of the algorithm. This makes sense since the first query is the most complicated one. It asks for a big regular expression of every prefix by or followed by every possible suffix. The measurements are presented in a histogram in figure 3.2 and in table 3.2. Note that, to reduce runtime, this time the initial prefix size selected was 4 as $2^4 = 16 > 2 \cdot 7$, which means there are at least two prefixes corresponding to each state of the largest state machine we want to learn. This means the set of prefixes can be split into at least the number of states of the machine, and there are (likely) going to be enough prefixes in each state for each possible transition, so we can construct an automaton (since

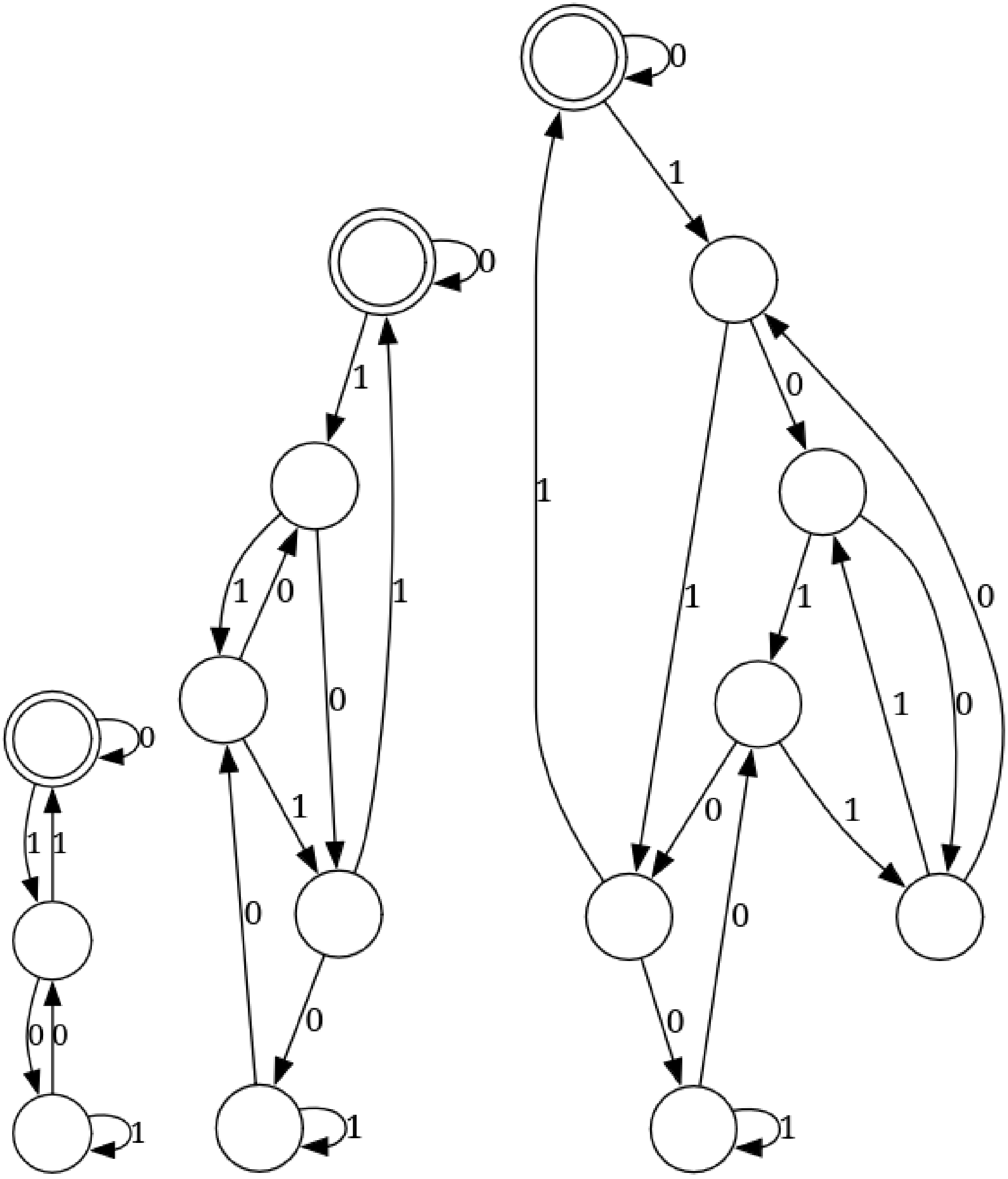


Figure 3.1: DFAs learned by the PoC for binary multiples of 3, 5, and 7 (left to right).

the binary alphabet has 2 characters, so 2 possible transitions).

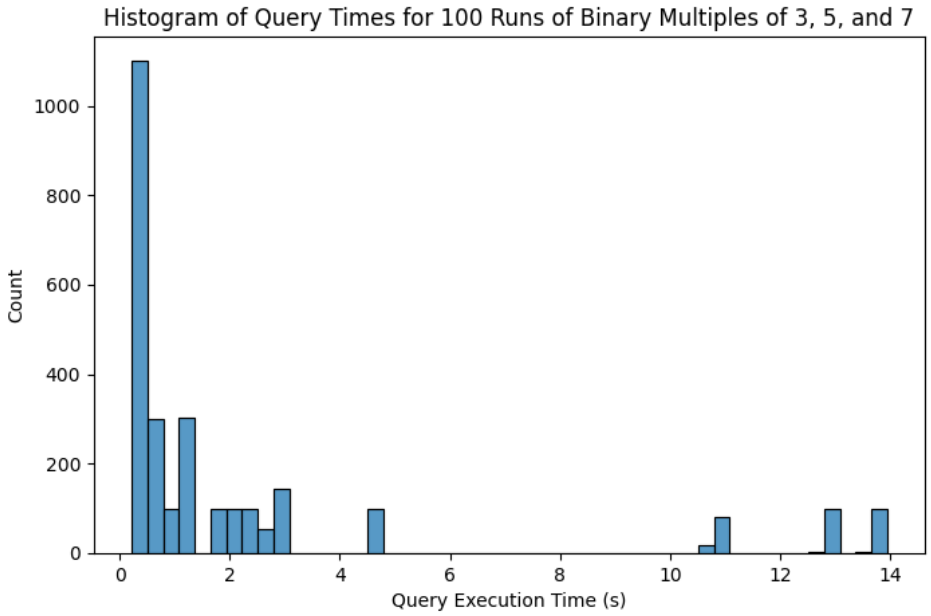


Figure 3.2: Histogram of Query Times.

As we can see, only 300 queries take longer than 10 seconds (the first three queries each time), and the majority of queries take less than 5 seconds, with almost all queries except the first ones taking under 2.5 seconds. Therefore, we can probably improve the runtime of the algorithm by a huge factor if we can avoid the first query.

Query #	Binary Multiples of 7 (s)	Binary Multiples of 5 (s)	Binary Multiples of 3 (s)
1	13.7895	12.9151	10.8325
2	1.0608	1.2506	0.6715
3	2.3262	3.0160	4.7305
4	2.1674	2.7932	0.6177
5	1.1939	1.8772	0.6641
6	1.1806	0.3589	-
7	0.2999	0.3550	-
8	0.2250	0.3569	-
9	0.2311	0.2973	-
10	0.2312	-	-
11	0.2331	-	-
12	0.2880	-	-
13	0.2323	-	-
Total	**23.4591**	**23.2202**	**17.5164**

Table 3.2: Mean execution times of queries over 100 runs.

Notice that 7 has two more states than 5, and it has four more queries than 5, which itself has two more states than 3 and four more queries than 3. The algorithm makes two queries per state, as it queries once to generate it from a previous merged state and then once more to verify it can't be split further.

3.6.3. Simple Improvements

Based on the observations of the previous section, we can make some improvements in the algorithm. One simple improvement is to skip the first query. We can do this by asking the membership status

of each prefix of the initial prefix set (the initial equivalence class) and splitting based on whether it is accepting or rejecting. This is equivalent to searching only for ε as a distinguishing suffix and dropping the $*$ part of the regex. In the event that the initial class has only accepting or only rejecting prefixes, this won't yield any improvement; this is a rare occurrence. The new execution times of the queries are presented in table 3.3 below. Moreover, it shows the new total times of all queries and the speedups compared to prior to the improvement. As we can see, this is a significant improvement, making the algorithm almost 2.5 times faster.

Query #	Binary Multiples of 7 (s)	Binary Multiples of 5 (s)	Binary Multiples of 3 (s)
1	2.2886	3.0246	4.7039
2	2.1455	2.7856	0.6096
3	1.2185	1.9096	0.6578
4	1.1815	0.3622	1.2360
5	0.2991	0.3568	-
6	0.2249	0.3564	-
7	0.2311	0.3013	-
8	0.2288	1.3058	-
9	0.2303	-	-
10	0.2798	-	-
11	0.2298	-	-
12	1.1162	-	-
Total	**9.6741**	**10.4024**	**7.2072**
Speedup	2.42	2.23	2.43

Table 3.3: Mean execution times of queries over 100 runs.

3.7. Comparison with Active and Passive Learning At This Point

3.7.1. Comparison with Passive Learning Algorithms

In order to compare this approach with established methods of passive learning, I selected the most common state-of-the-art algorithm, state merging with the EDSM criterion, and in particular its efficient C++ implementation in the FlexFringe framework [14]. I recorded the runtimes of each method over 100 runs in the same datasets of multiples of 3, 5, and 7. In order to test scaling, I run in various dataset sizes. In particular, 1500 traces, 15000 traces, 150000 traces, and 1.5 million traces. The results are presented in a log-log graph in figure 3.3.

Notice that both algorithms are linear in terms of the dataset size. In all cases, the described algorithm is faster than EDSM, despite being implemented in Python. Moreover, EDSM runs out of memory after about two and a half minutes in the case of 1.5 million traces and fails, while our method expectedly has no issues due to using the database.

3.7.2. Comparison with Active Learning Algorithms

Moreover, despite the fact that active methods do not use a dataset, we can still compare with them. Since the machines of the datasets are simple in checking membership, active learning methods tend to be very fast. However, their disadvantage is that they often make large numbers of queries, and in cases where queries are slow, they can be slower. As such, it is reasonable to compare the number of queries between methods. So, I run the classic L* algorithm [2], 100 times for binary multiples of 3, 5, and 7. I used the AALpy library in Python [8] and made it learn a function which accepts a binary string and responds whether it represents a multiple of a given number. For each run, I calculated the mean total number of queries. As we have seen already, the described method performs about twice as many queries as there are states in the final machine, every time. The results are in table 3.4.

As we can see, the proposed method makes significantly fewer queries than L*. In cases when the system under learning is complicated, and queries take a long time, it is expected that our method will be significantly faster.

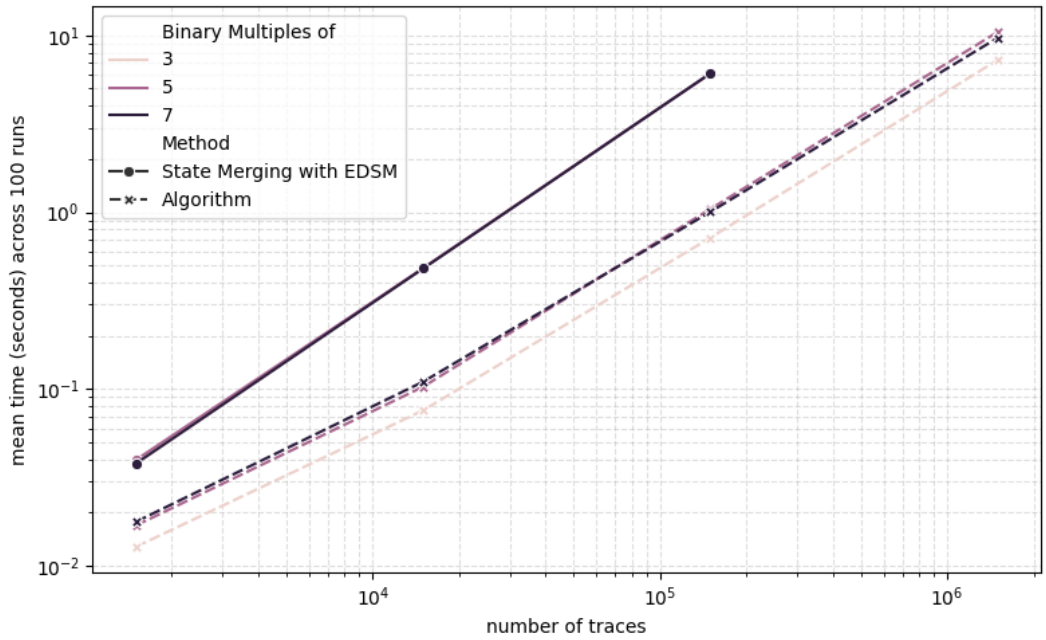


Figure 3.3: Mean time of EDSM vs the Split-Explore Loop (labeled algorithm) over dataset size, log-log scale.

Binary Multiples of	Mean Number of Queries of L*	Number of Queries of our Method
3	160.65	4
5	273.03	8
7	391.77	12

Table 3.4: Mean number of queries over 100 runs.

4

Split-Explore Loop

Research Question How can we extend the algorithm so it automatically searches for new prefixes when it doesn't have enough to properly learn the automaton?

An issue with the algorithm developed so far has been that the initial set of prefixes is significant for finding the correct DFA. If it doesn't have at least one per state, it will miss these states. At the same time, some states might require long traces to reach. Since our mechanism up to now was to add all prefixes of a given length, which is exponential, this is infeasible for such cases. Instead, we would like for the algorithm to automatically explore traces and figure out whether it should add some useful longer ones, without adding all possible ones of equal length.

4.1. Idea

Crucially, we need a method that, given a prefix p, can detect in which of the classes discovered thus far it should be placed. A simple proposal is to pretend the prefix was there all along and somehow trace its path in the various classes. For that, we need to keep track of the distinguishing suffixes discovered, which leads us to the following data structure.

4.2. Distinguishing Suffix Tree

A *Distinguishing Suffix Tree* (DST) consists of nodes corresponding to equivalence classes and edges corresponding to suffixes that distinguish these classes. In Myhill-Nerode terminology, if $x \in C_1, y \in C_2$ and $z : x \cdot z \in L \iff y \cdot z \notin L$, then z being a distinguishing suffix for these classes could be an edge between them in the tree, $C_1 \xrightarrow{z} C_2$.

Examples of such a tree can be seen in figure 4.1. It is a DST for the DFA of decimal multiples of 3. A distinguishing suffix for classes 0 and 1 is ε. For the annotation next to the distinguishing suffix, see the next section.

4.3. Annotated Distinguishing Suffix Tree

An *annotated distinguishing suffix tree* is a directed DST that has some additional information useful for adding newly discovered prefixes to the equivalence classes.

In figure 4.1, annotated next to the distinguishing suffix is some additional information on the membership, which will be of use later when we use the trees to add new prefixes. It reads, in accordance with the previous notation, $z = \varepsilon$ and $y \cdot z \in L$ is false.

It is directed in the direction of the splits. For instance, if we split C_2 from C_1 with z, the edge should be $C_1 \xrightarrow{z} C_2$. The reason is that when simulating the new prefix being there from the start, we will start by placing it in the initial class and then moving it toward the new classes if required.

It additionally contains the information of whether the prefixes $y \in C_2$ of the destination node are in the

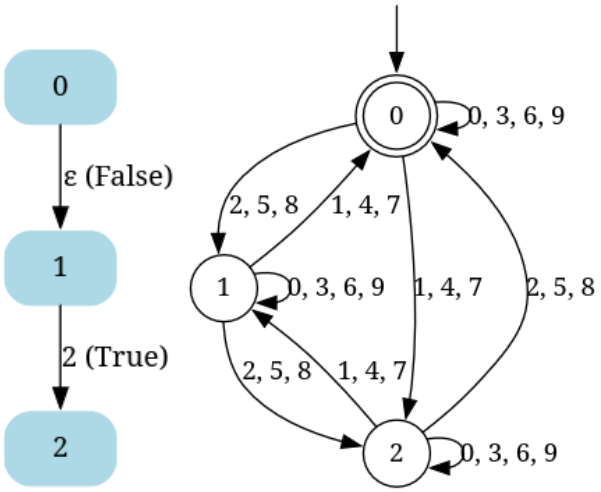


Figure 4.1: Distinguishing suffix tree for decimal multiples of 3 (produced by the implementation).

language when extended with the distinguishing suffix or not $(y \cdot z \in L)$. This is required because when adding the new prefix p, we can test $p \cdot z$ and move it to one of the two equivalence classes accordingly.

It also contains information on the order in which the nodes were added to the tree. This is useful because when simulating the new prefix being there from the start, in case multiple distinguishing suffixes match, we should prefer moving it to the older one (because it would have moved there before the next matching split).

4.4. Adding New Prefixes

Now we can use the information stored on the annotated distinguishing suffix tree to add a set of new prefixes by essentially performing membership queries of the prefixes, followed by each suffix in the path.

It's better to operate in batches instead of each new prefix individually because that makes fewer queries to the database, and they may take less total time to complete. Note that this contains some redundant information; however, both the number of prefixes and suffixes is usually small, and in the worst case, every combination needs to be checked. The pseudocode is presented in algorithm 7.

Algorithm 7 Method that identifies the classes of a set of prefixes and adds them to these classes.

```
1:  procedure AddPrefixes(P, T)
2:      S ← T.suffixes
3:      Q ← MembershipDB(P × S)                          ▷ Efficient batched query
4:      for p ∈ P do
5:          C ← T root
6:          while C has children in T do
7:              for all (C', z, label) ∈ T.children(C) in order of appearance do
8:                  if Q[p · z] = label then
9:                      C ← C'
10:                     break
11:                 end if
12:             end for
13:         end while
14:         C ← C ∪ p
15:     end for
16: end procedure
```

4.5. Populating a Class

An interesting additional operation we can do with the annotated distinguishing suffix tree is to select a target candidate class and retrieve prefixes that, according to our model so far, should belong to it.

Algorithm 8 works by defining a desired path through the tree, blocking alternative paths and children, and then gathering the suffixes of the paths for querying the database about prefixes.

A key method is SuffixQueryDB(S^+, S^-, n) that takes a positive set of suffixes S^+, a negative set of suffixes S^-, and a number n, and returns n prefixes that, when prepended to all suffixes of S^+, are accepted, and when prepended to all suffixes of S^-, are rejected. Its SQL implementation is in appendix A.

Algorithm 8 Method that, given a class, pulls from the database prefixes that should belong to it according to the model so far.

procedure $\mathrm{Populate}(C, n, \mathcal{T})$
    Collect all ancestor edges on the path from $T.\mathrm{root}$ to C in $\mathcal{T}$
    For each edge labeled (z, label), record the suffix z with the label ($+$ accepted or $-$ rejected)
    Collect children of C, with opposite label
    Collect bifurcations in the path that predate C's ancestor, with opposite label

    $S^+ \leftarrow \{s \mid (s, +) \text{ was recorded}\}$ ▷ Suffixes that must evaluate to accepting
    $S^- \leftarrow \{s \mid (s, -) \text{ was recorded}\}$ ▷ to rejecting

    $P \leftarrow \mathrm{SuffixQueryDB}(S^+, S^-, n)$
    $C \leftarrow C \cup P$
end procedure

4.6. Worked Out Example in a Simple Tree

To better illustrate these two methods, this section contains an easy example on the annotated DST of Figure 4.1. We will call AddPrefixes($[3, 4, 5]$) and Populate(C_1).

AddPrefixes($[3, 4, 5]$**)**
We query about each trace in $\{3, 4, 5\} \times \{\varepsilon, 2\} = \{3, 4, 5, 32, 42, 52\}$.

- **add 3**: Starting from C_0, $3 \cdot \varepsilon = 3$ which is accepted, so 3 remains in C_0 (figure 4.1).
- **add 4**: Starting from C_0, 4 is rejected, so move to C_1. 42 is a multiple of 3, it's accepted, so move to C_2 (figure 4.1).
- **add 5**: Starting from C_0, 5 is rejected, so move to C_1. 52 is not a multiple of 3, it's rejected, so it remains in C_1 (figure 4.1).

Therefore, we found that $3 \in C_0, 4 \in C_2, 5 \in C_1$

Populate(C_1**)**

- **Ancestors of** C_1: $(C_0, \varepsilon, -)$.
- **Children of** C_1: $(C_2, 2, +)$.
- **Bifurcations in the path to** C_1: None. (This tree is linear, so this does not apply. In other trees, it blocks traces that would follow alternate paths and end up in other classes.)

So $S^+ = \emptyset, S^- = \{\varepsilon, 2\}$.

From SuffixQueryDB, we will obtain traces that are not multiples of 3 (rejected with ε) and when appended with 2 are also not multiples of 3. This includes the number 5 of the previous example ($5 \cdot \varepsilon$ and 52 were rejected) and also numbers of the form $3n + 2$.

4.7. Split-Explore Loop

We can use these two new methods to include new prefixes in our classes dynamically while the algorithm is running. As such, the new loop will contain two phases: one splitting classes as before and one exploring for new prefixes that might be useful (algorithm 9).

Algorithm 9 Improved Main Loop with Class Population and Prefix Expansion

Require: Database connection, C_0: initial equivalence class, n: number of new prefixes per iteration
Ensure: DFA constructed from final equivalence classes

$\mathcal{T}.\text{root} \leftarrow C_0$ ▷ Distinguishing suffix tree initialization
$\mathcal{P} \leftarrow \{C_0\}$ ▷ Classes awaiting processing/splitting
$\mathcal{F} \leftarrow \emptyset$ ▷ Finalized equivalence classes
while $\mathcal{P} \neq \emptyset$ **do**
  $C \leftarrow \mathcal{P}.\text{pop}()$
  $\text{Populate}(C, n, \mathcal{T})$ ▷ Enrich class with n new traces from DB
  $\mathcal{S} \leftarrow \text{SplitClass}(C)$ ▷ Either $\emptyset$ or $\{C, C'\}$
  if $\mathcal{S} \neq \emptyset$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{S}$
    $\mathcal{T}.\text{children}(C) \leftarrow \mathcal{T}.\text{children}(C) \cup (C', z, \text{label})$
    ▷ the distingusihing suffix z, and label can be returned by SplitClass
  else
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{C\}$
  end if
  ▷ Expand frontier using missing transitions
  $P \leftarrow \left\{ x \cdot a \;\middle|\; \begin{array}{l} x = \text{representative}(C),\ C \in \mathcal{P} \cup \mathcal{F}, \\ a \in \Sigma,\ \text{no transition defined for } (C, a) \end{array} \right\}$
  $\text{AddPrefixes}(P, \mathcal{T})$
  if a prefix was added to a class $C \in \mathcal{F}$ **then**
    $\mathcal{F} \leftarrow \mathcal{F} \setminus \{C\}$ ▷ A prefix was added to a finalized class, now it might require further splitting
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{C\}$ ▷ So bring it back to processing
  end if
end while
return $\text{BuildDFA}(\mathcal{F})$

The most logical place for Populate($C, n, \mathcal{T}$) is right before splitting, as then having more prefixes leads to a wider search for distinguishing suffixes and possibly the discovery of more states that would have been missed.

Additionally, I decided to use the AddPrefixes($P, \mathcal{T}$) method to add the missing transitions between classes after a split. This guarantees that BuildDFA($\mathcal{F}$) (algorithm 6) will not be missing any transitions when it moves to build the DFA. It also serves in the correctness proof in Chapter 7.

4.7.1. Detecting Missing Transitions and Adding Them

We can easily identify the missing transitions for each class and add them. Given a class C look for each symbol $a \in \Sigma$ for which, if we extend all prefixes of that class $p \in C$ so they become $p \cdot a$, we cannot find another class C' (or class C itself) such that $p \cdot a \in C'$. In that case, we don't know where the transition from C with a is supposed to go.

Once these symbols are identified, we only need to select a representative $x \in C$ and add $x \cdot a$ through the tree. This new prefix will end up in some other class C' (or class C itself), and thus it will now define the transition $C \xrightarrow{a} C'$.

Analysis of this follows in the chapter on correctness, Chapter 7.

4.8. Query Variant Version

4.8.1. Reasoning

There are several reasons that motivate the development of another variant of the algorithm. As we will see later in the Results chapter, the Populate() method and the query are currently computationally expensive. Moreover, in the Correctness chapter, certain situations arise in which the pumped suffix might not work. Additionally, the variant presented here has a full correctness proof demonstrated in the Correctness chapter. Finally, this version might be easier to optimize since the query is a specific task.

4.8.2. Description

The main loop is identical to algorithm 9 with two differences. SplitClass(C) in line 7 operates slightly differently. Instead of using the regex with the pumped suffixes of algorithm 3, it uses a set of prefixes P and matches all traces of the database that begin with any of the prefixes in P. Then, as with the regular query, it selects the shortest suffix and returns all prefixes that have that suffix and the requested label.

It can be thought of as using that regex with any possible suffix, `^(`*prefixes*`)(.*)$`. However, it is faster to implement the query with a `CROSS JOIN` in SQL, and there might be even faster ways of implementing this operation on appropriate data structures.

Algorithm 10 Variant of SplitClass

```
1:  function SplitClassVariant(C)
2:      P ← prefixes(C)
3:      C_opp ← QueryVariantDB(P, label ≠ C.label)        ▷ Strings with opposite label
4:      C_same ← C \ C_opp
5:      if C_opp ≠ ∅ and C_same ≠ ∅ then
6:          return {C_opp, C_same}
7:      else
8:          return ∅
9:      end if
10: end function
```

The second difference is that the Populate($C, n, \mathcal{T}$) call of line 6 is not needed anymore. The reason is that the query of SplitClass(C) now looks for any possible suffix, so prefixes are not as important to finding a distinguishing suffix. Additionally, as we will see in the correctness proof of this version in Chapter 7, we only need to add the missing transitions to discover all states.

4.9. Demonstrations

To test the ability of the split-explore loop to autonomously discover new states, an ideal language is one where the k-th character is a specific character. I chose binary strings where the k-th character is 0.

The simple split loop requires initialization with prefixes of length k to correctly identify the automaton (since the only accepting state is length k deep). The split-explore loop correctly identifies the DFA in each case, even when initialized with prefixes of length just 1 (only the alphabet). The population parameter was set to 5 new prefixes before each split.

The resulting DFA is depicted on the left in Figure 4.2. Note that because of the way the dataset was generated, no traces start with 0. So any trace starting with 0 going to the junk state is technically correct.

4.9.1. Long DFAs

The split-explore loop can also discover long DFAs now, as we can see in figure 4.2.

Note that the right DFA accepts traces whose 16th symbol is 1 and whose 32nd is 0. This is because the longest trace in the dataset had 21 symbols, so there was no example of a 32-symbol trace in the dataset, which was rejected. So the backward transition from state 14 to 0 (and missing the junk state)

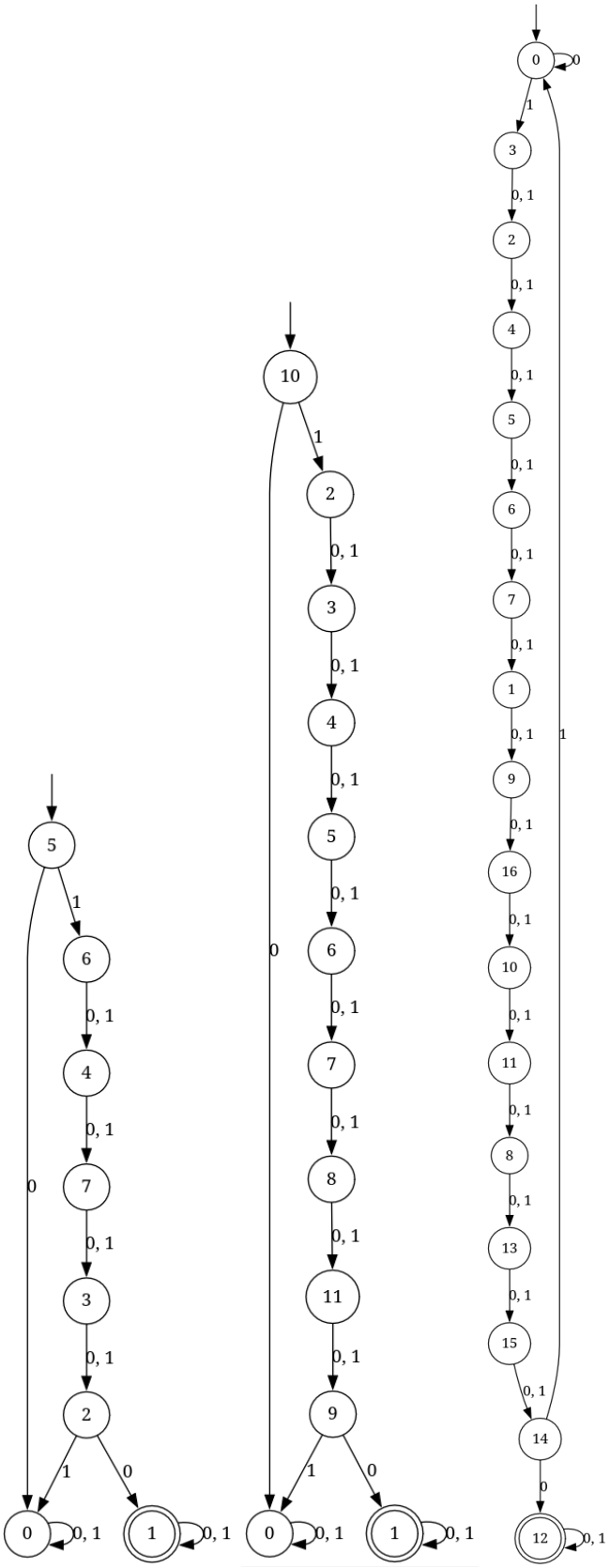


Figure 4.2: DFAs constructed by the algorithm for a dataset where the 6th (left), 10th (middle), and 16th (right) binary digit was 0.

is technically correct.

4.10. Discussion: Similarities and Differences with L*

The split-explore loop exhibits similarities with the L* algorithm. It also uses a (batched) membership query. The regex query of SplitClass can be considered analogous to the equivalence oracle, though it answers a different question: whether a state is correct rather than whether the whole model is correct.

On the other hand, it has some key differences. The populate method has nothing similar in L*. L* uses a table to store suffixes, while this algorithm uses a distinguishing suffix tree.

A conclusion from this is that this database-assisted algorithm is also an in-between approach between active and passive learning. It does not interact with the system, and the queries it asks are not answerable by interaction anyway. However, it more closely resembles the L* structure of querying, filling in a data structure, and proposing models.

5

Evaluation Methodology

In this chapter, I describe the experiments I set up to compare the algorithm to passive and active learning. Results are demonstrated in the subsequent two chapters.

There are several properties of the algorithm we would like to test. First, we want to compare its performance against passive learning methods in terms of time and against active learning algorithms in terms of the number of queries. We also want to compare the number of states discovered for each algorithm as a baseline for their correctness. Moreover, we want to see the differences between the two variants of the algorithm.

5.1. Datasets

5.1.1. Generated

Two kinds of datasets were generated to evaluate the algorithms.

The binary multiples of n, the dataset already used in the third chapter, was a natural candidate for a synthetic large dataset because of its ease to generate. Additionally, one easily sets the size of a machine by the number n when it is not even, and these machines feature complex transitions. As they are transition-rich and state-rich, they offer good baselines for both performance and model correctness.

Binary sequences where the k-th bit is 0 are the other dataset used. These evaluate how deep the model can reach when exploring DFAs. Note, they have $k + 2$ states (k to reach length k, 1 for ε, and 1 junk state, see figure 4.2).

5.1.2. Abbadingo

In addition to the generated datasets, I used the random DFAs from the Abbadingo challenge for some experiments. The Abbadingo datasets were originally created for the Abbadingo DFA learning competition [7], but they can also function as a good benchmark for passive learning algorithms.

The Abbadingo problem set consists of problems of varying difficulty, each generated by an unknown DFA. In each problem, there is a training set of labeled examples (positive and negative strings). This dataset was generated with a random walk through the target DFA. The datasets vary in:

- Target DFA states, ranging from $\sim$60 to $\sim$500,
- Target DFA depth, ranging from 10 to 16,
- Number of strings in the training set, ranging from 1521 to 115000.

In contrast to the generated datasets, these do not reach scales of millions of traces that would make simple passive learning approaches fail, as we saw previously.

The Abbadingo challenge also provides a test set for each train set to evaluate the accuracy of a predicted DFA. However, I did not use this because (as we will see in a following section outlining

the experiments) the main aim of the experiments with these datasets was to uncover how the split-explore loop algorithm operates under different target DFA conditions, which is not visible through a simple accuracy score.

On the other hand, the target DFA number of states and depth are useful information for these random DFAs and can indicate which conditions aid or impede the algorithm.

5.2. Parameterizations

The values of the parameters selected for the experiments and the reasoning behind them are as follows.

For multiples, the numbers chosen were 15, 47, 123, and 1155. 15 can be constructed as a combination of the simplest (non-trivial) binary machines of 3 and 5. 47 is a prime number offering a unique machine in the reasonable size of $\sim$50 states. 123 is a slightly larger machine in the range of $\sim$100 states. 1155 is a combination of the 4 first simplest prime machines ($1155 = 3 \cdot 5 \cdot 7 \cdot 11$), and it is in the order of $\sim$1000 states.

For k-th bit sequences, the numbers chosen were 6, 10, and 14. 6 is a mid-range depth that accurately checks reasonable model exploration depths. 10 is longer, hiding the accepting state behind longer sequences and thus needing greater exploration depths. 14 is a big depth.

The number of traces I tested ranged from 10 thousand to 1 million. These sizes are the interesting region because it is right where passive learning runs out of RAM.

Representations of the expected DFAs for the multiples of 15 and 47 datasets can be seen in figure 5.1. For the 6th and 10th is 0, they are the first two DFAs of figure 4.2.

5.3. Experiments

5.3.1. Algorithmic Parameters

Because the larger multiples datasets took a long time and since there was little variance in the resulting values (as I knew from the initial experiments), I chose to run each dataset 20 times instead of 100.

In the graphs of the following chapters, I chose “base” to signify the split-explore loop with the pumped regex and “no suffixes” for the variant version that only looks for any suffix.

Both variants were initiated with just the alphabet as prefixes (length 1). Additionally, base used $n = 5$ in the Populate($C, n, \mathcal{T}$) query because it offers a reasonable trade-off between accuracy and speed (higher populate adds many prefixes and slows down all queries).

5.3.2. Experiment Descriptions

I run both variants of the split-explore loop (implemented in Python) on the datasets of binary multiples of n and binary sequences where the k-th character is 0. I also run the FlexFringe implementation of EDSM as a benchmark for passive learning methods, and the AALpy implementation of L* as a benchmark for active learning methods, both with default parameters.

The datasets for the split-explore loop were generated and inserted into a PostgreSQL database. For FlexFringe, they were generated in `.dat` files in the abadingo format it requires. For L*, I implemented simple functions returning true for traces of the language and false otherwise.

In the two experiments for the two datasets, five metrics were recorded for each algorithm (if they are applicable). Number of states discovered, total execution time, number of oracle queries, number of learning queries, and number of prefixes used.

Additionally, I run another experiment recording the time each query took in both variants of the split-explore loop in the heavier of the two datasets, the binary multiples. This will help analyze hotspots of the algorithm and potential improvements.

Finally, as an additional investigation for the correctness chapter, I evaluated the variant that looks for all possible suffixes on the Abbadingo dataset. I chose that variant because, as we will see in the proper place, it performed better in terms of output DFAs in the previous experiments. The evaluation

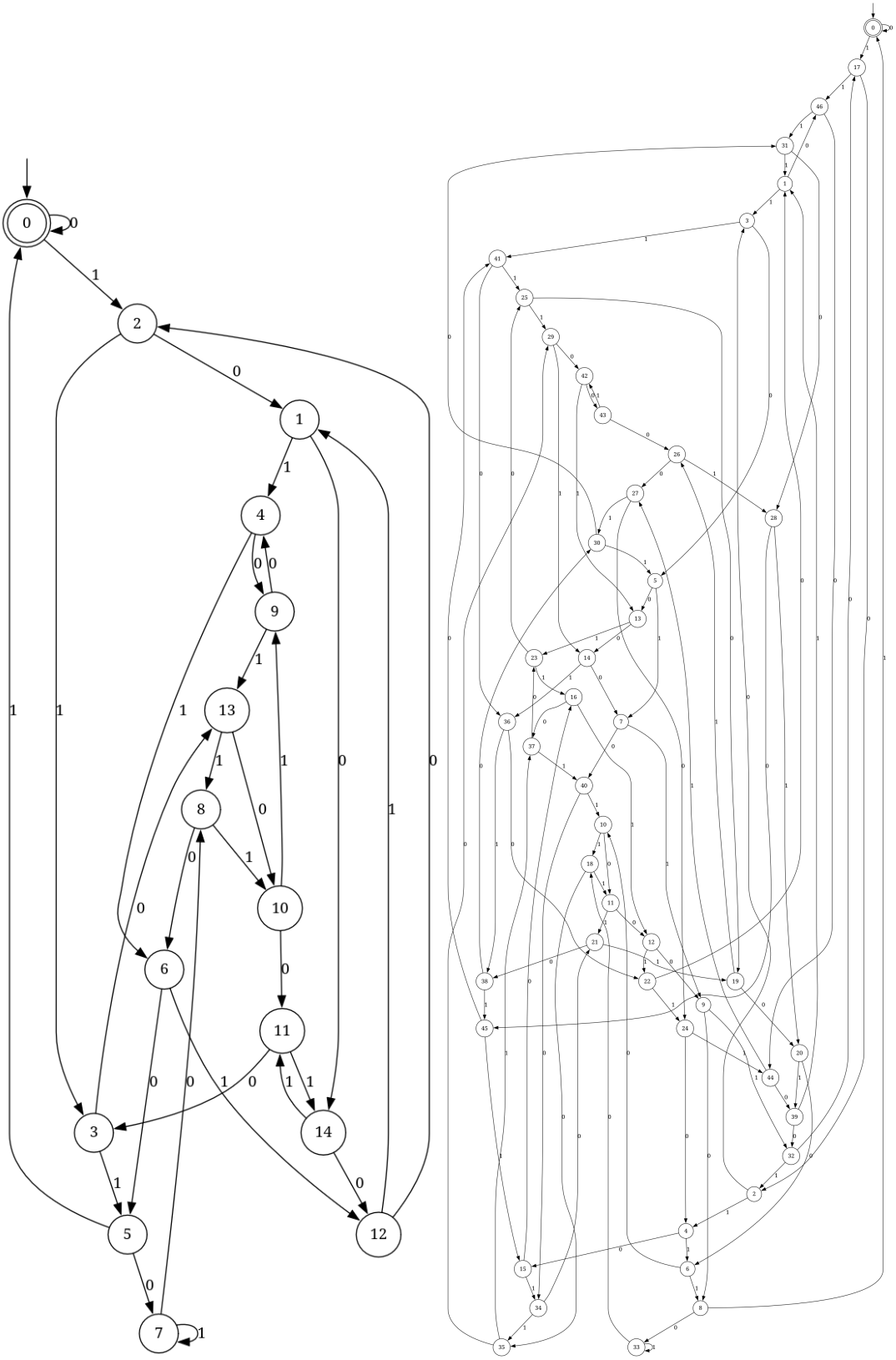


Figure 5.1: Correct DFAs for binary multiples of 15 (left) and 47 (right). Produced by the split-explore loop.

was in terms of states discovered in the random DFAs of the Abbadingo dataset, which might be a bit more realistic compared to the generated datasets. Further discussion is in section 7.4.

6

Results & Comparisons

Research Question How does the new approach compare with passive and active learning?

This chapter answers the above research question by presenting and analyzing the results of the experiments described in the previous chapter. Because all the results have low variance, the means alone are quite good representations.

Throughout this chapter, certain naming abbreviations are used, especially in the graphs. “base” denotes the first variant of the split-explore loop of Chapter 4 that uses a regular expression when searching for prefixes. “no suffixes” denotes the second variant that does not use a regex and instead explores the whole database. Perhaps a better name for it would have been “all suffixes” as it searches for every possible suffix of the prefixes of a given equivalence class. I initially named it “no suffixes” because the main query does not use suffixes; only prefixes. Changing all the graphs now proves to be a bit difficult.

Additionally, “muls” denotes the dataset of multiples of a number (i.e., 100k muls 123 means the dataset of multiples of 123 of size 100k samples). The datasets where the k-th bit is 0 are denoted as “10th 100k” (meaning the 10th bit is 0 and it is of size 100k samples).

6.1. VS Passive Learning (Time)

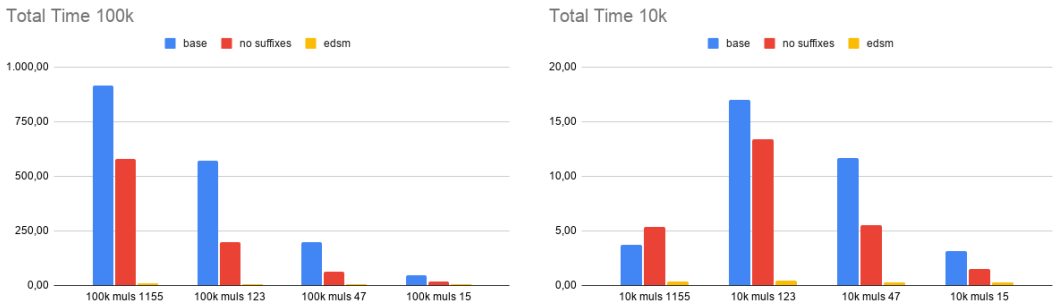


Figure 6.1: Total time for each method in multiples datasets.

As one can see in figure 6.1, when comparing the split-explore loop with the FlexFringe implementation of EDSM in binary multiples, it is significantly slower. However, at a million traces, EDSM runs out of RAM and crashes after about 60 seconds. These methods, as expected, do not, and in fact, as we will see later, due to their small number of prefixes in memory (< 2000), they can probably scale well beyond that.

Remarkably, the no suffixes method is faster in almost all cases; this is investigated in the following section. As expected, increasing the size of the dataset makes the algorithms slower because the

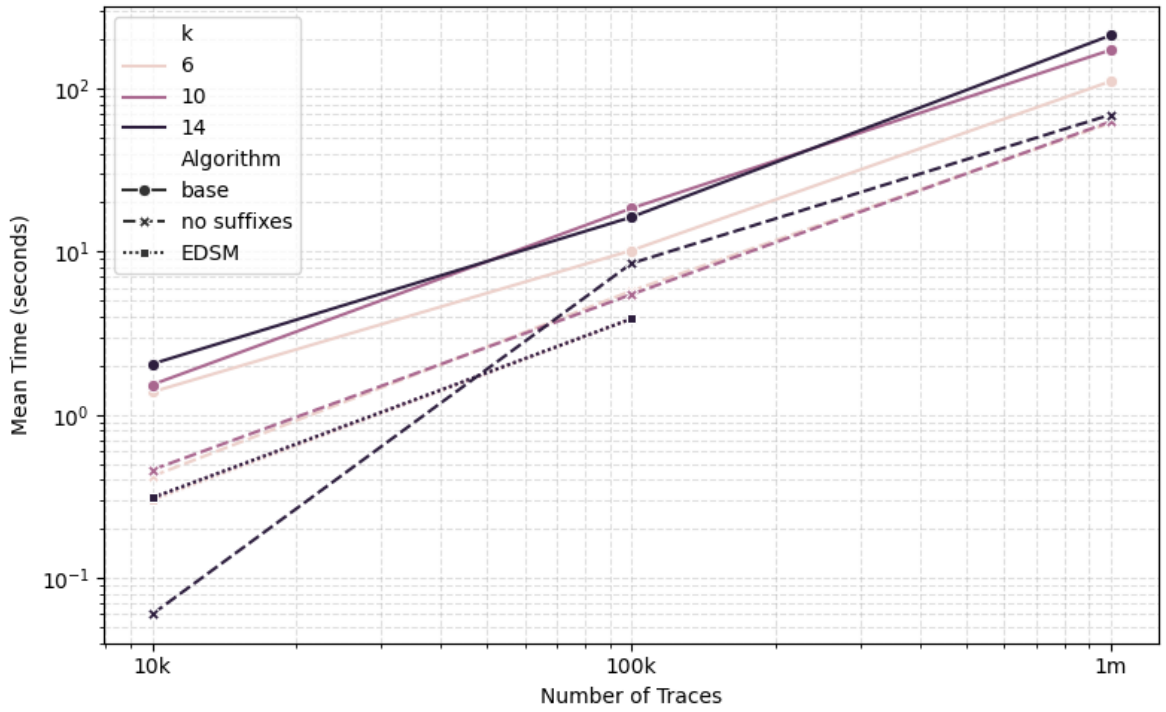


Figure 6.2: Time vs dataset size for the split-explore loop and EDSM (log-log scale).

queries become slower through a bigger database. Increasing the number of states makes the algorithms slower because it increases the number of iterations of the main loop, which splits off one class at a time.

In figure 6.2, things look better for the split-explore loop. The time difference is smaller, though EDSM is still the fastest. Again, EDSM runs out of RAM at 1 million traces and does not produce a result after about 60-70 seconds.

The no suffixes variant is again faster, though at 10k traces in “14th is set to 0” it fails to detect a split, and it outputs an incorrect DFA with only one state. This explains why it appears to be the fastest method there. This is rectified with bigger dataset sizes.

An interesting observation is that, like before, the plot is log-log, and we can thus see the split-explore loop also scales linearly with the dataset size, despite its complexity. This is great, as the primary aim of this algorithm is scaling.

As we will see in the correctness Chapter, where the number of states discovered in each dataset will be presented, on average, both variants very slightly outperform EDSM in getting correct results in the k-th bit datasets at the same size. Moreover, running on the datasets of size 1 million, the split-explore loop gives an even better DFA while as we can see here EDSM cannot scale to that size and use more data to improve its DFA.

6.2. Time by Query Type for the Split-Explore Loop

I collected data for the average time each type of query took for each iteration during an execution of the algorithm. I then repeated this 20 times and re-averaged the results to get an idea of how costly each query is. The data is presented in figure 6.3.

As we can see in Figure 6.3, the query of all suffixes is significantly slower than the pumped query. The reason for that is that the query is searching the entire database, and as expected, it’s even worse for larger databases. In contrast, the pumped suffix significantly limits the traces that are going to be processed after the regex to find a counterexample suffix.

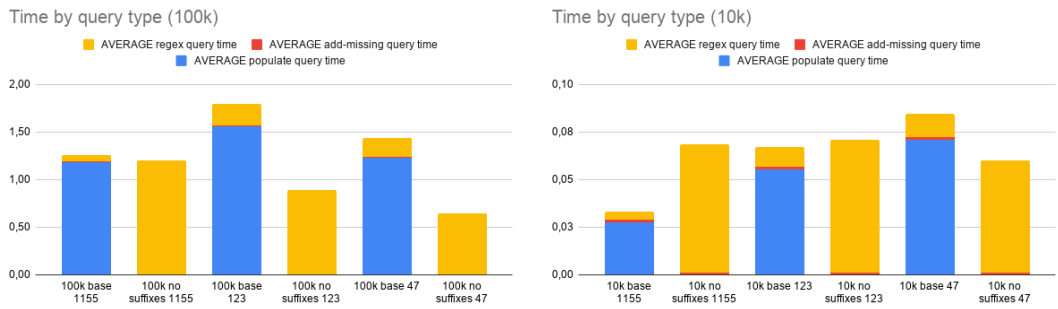


Figure 6.3: Time by query type for the two variants in various datasets.

What was slowing down the “base” version was actually the populate query. That query is complicated, too, so it searches the entire database before performing an intersection. For this reason, the limit factor does not make it faster. It has to be limited after the intersection, but the intersection needs the entire database of the two queries to be performed. Perhaps pumping some prefixes could make it faster.

We can also see that adding missing transitions has almost no cost in each case. The reason for that is that the query is much simpler. It has no post-processing after the regex is run, and the matching traces are few and processed in batches (the prefix followed by each of the distinguishing suffixes, which is in the order of the number of states of the machine).

An interesting conclusion from this is that the main optimization points for this algorithm should be the populate query and the “no suffixes” variant query for the variant.

6.3. VS Active Learning

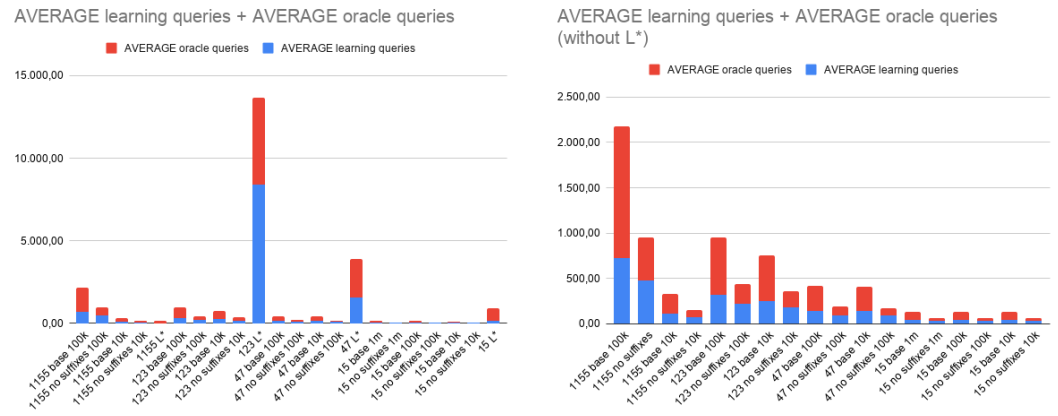


Figure 6.4: Average total number of queries per method and dataset (binary multiples). On the right without L*.

Again, we compare with active learning methods in terms of the number of queries to oracles. In order to count the number of queries for our methods, we can utilize the L* similarity described in Chapter 4. As such, we can consider the regex query as equivalent to the oracle query of L*, and the `add_missing` and the `populate` queries are part of the learning queries, equivalent to the membership queries of L* (though populate does not exactly correspond to either category).

As we can see, L* makes a huge number of queries, much more than the two variants. For that reason, the results are plotted again without L* on the right. Note that, as we will see later, in 1155 L* finds an incorrect automaton with only 2 states. This is the reason it appears to have such a small number of queries in that dataset.

Additionally, we can see as expected that “no suffixes” makes fewer queries, which is expected because

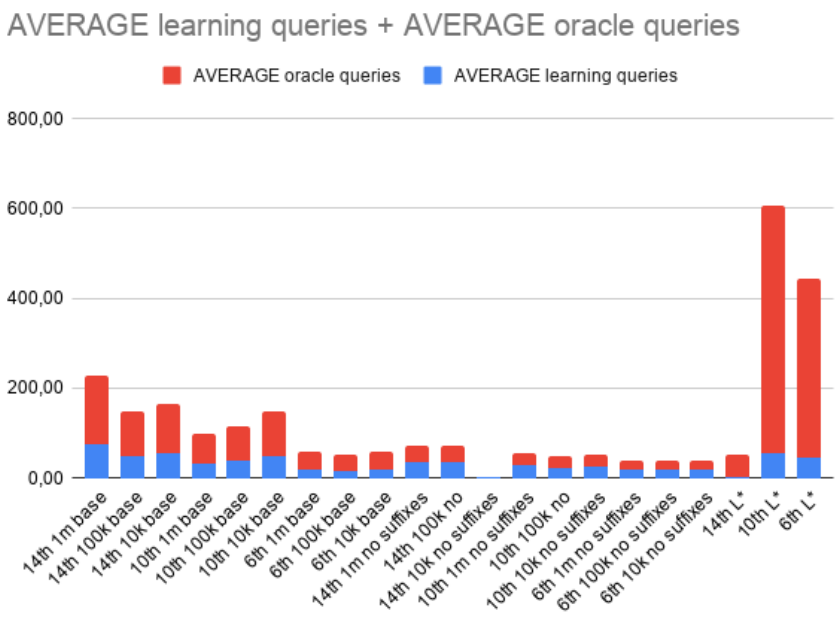


Figure 6.5: Average total number of queries (k-th bit is 0 datasets).

it does not use populate at all. However, it also makes fewer oracle queries, which implies it finds the automaton with fewer iterations. The reason is that the “no suffixes” query finds a distinguishing suffix more easily since it only matches prefixes, while the pumped regex might more often initially not find a distinguishing suffix and finalize the state only to find later that a new prefix added can split it (lines 16-19 in algorithm 9).

As expected, a larger dataset requires more queries, and a larger machine also requires more queries. The reason is that the bigger the dataset, the easier it is to find counterexamples and more states, so the algorithms make more iterations (this will also be shown in the section about the number of states).

In Figure 6.5, the same results are depicted for datasets of the k-th bit set to 0. Overall, the conclusions are similar.

One observation is that the ratio now is more skewed towards oracle queries for both algorithms compared to the multiples.

Note that, in the 14th bit set to 0, L* produces an incorrect DFA with only one state, too, and hence the small number of queries. This is also true when running the “no suffixes” variant with 10k samples in the same dataset.

6.4. Number of Prefixes

Finally, another interesting metric to investigate is the total number of prefixes among equivalence classes used to derive the automaton for each of the two methods. The reason is that more prefixes make more complicated queries and slow down the algorithm. Using fewer prefixes to derive the same machine means more efficient use of prefixes and queries. Fewer prefixes also allow for higher scaling as fewer items are stored in RAM.

As we can see in figure 6.6, the no suffixes variant in each dataset consistently uses significantly fewer prefixes than the base one. This was expected because the no suffixes variant does not use the populate query. The populate query adds 5 prefixes each time before splitting, which inflates the number of prefixes used. Perhaps a more sophisticated population query could target more useful prefixes better through heuristic methods.

We can see that for datasets up to 100k traces, these methods store about 200-1k prefixes in RAM. This

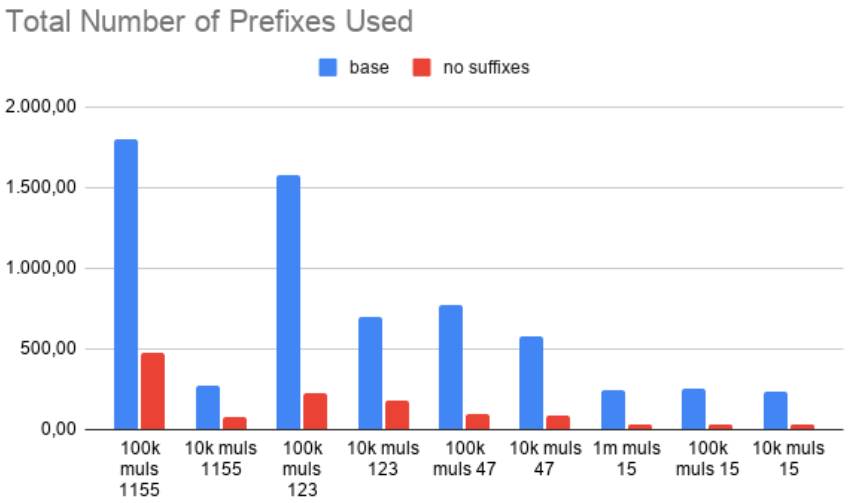


Figure 6.6: Total number of prefixes used by the two variants of the algorithm in various datasets.

gives a condensation of about 100-1000 times, rendering these algorithms highly suitable for massive datasets.

6.5. Summary of Comparisons to Passive and Active Learning

The experiments of this chapter demonstrate that the split-explore loop achieves scalability unmatched by passive learning methods while using significantly fewer queries than active learning methods.

Compared to passive learning, it is slower than EDSM due to the more complicated database queries. However, unlike EDSM, the split-explore loop is not constrained by available RAM and can scale to much larger datasets (1 million traces or more), and producing more accurate automata by using more data. Furthermore, the relation between time needed and dataset size is shown to be linear.

Compared to active learning, the split-explore loop makes orders of magnitude fewer queries.

Additionally, through the experiments, we found that the population query is the slowest, and it is slightly slower than the query variant that checks all suffixes in these datasets. Moreover, both variants use at most 2000 prefixes in RAM (in most cases, even less), achieving a $\sim$1000 times condensation of the dataset, allowing for very big scaling.

7

Correctness

This chapter investigates the correctness of the split-explore algorithm developed. It starts by showing some theoretical limitations of the base variant, and then moves on to prove the correctness of the "no suffixes" variant. Then it analyzes the results of the experiments regarding the number of states discovered.

7.1. Pumped Regex Blind Spot

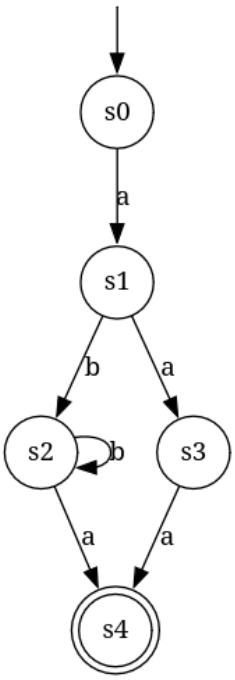


Figure 7.1: Enter Caption

Figure 7.1 demonstrates an interesting situation. If we assume every state has been discovered except s1 and s2, which have not been split yet, we will find that the pumped regex is unable to split them. Merged state s12 contains prefixes {a, ab, abb, abbb, …} (while s1 contains {a}, s3 contains {aa}, and s4 contains {aaa, aba, abba, …}). Constructing the regex in the usual way, we obtain `^(a|ab|abb|abbb|...)(b)*$`.

All the matched traces of this regex are either a, which is rejected, or end with b, and all are rejected because this automaton accepts only traces ending with a. Therefore, a distinguishing suffix will not be found. However, a distinguishes the prefixes a and ab, because aa is rejected while aba is accepted.

The pumped regex thus fails to split s1 and s2 because it has a blind spot. It is not known if this situation may or may not arise while the base variant of the algorithm is running; however, it indicates that a correctness proof for it might not be possible.

Potential mitigations are discussed in the future work section of the concluding chapter. In the meantime, I provide a correctness proof for the other variant.

7.2. Correctness Proof

Research Question Can we give a correctness proof?

If we assume that every state in the target DFA is reachable and that the database contains sufficient information, we can prove the correctness of the "no suffixes" variant as follows.

Before we can prove the main theorem, we need to prove the following helpful lemma.

Lemma Let P be a set of prefixes that lead to *exactly two* distinct states q_1 and q_2 of the DFA of a language L. Then $\mathrm{SplitClassVariant}(P)$ (algorithm 10) returns two non-empty subsets P_1 and P_2 such that all prefixes in P_1 reach q_1 and all prefixes in P_2 reach q_2.

Proof By the Myhill-Nerode theorem, since the prefixes in P lead to two different states, there exist $x, y \in P$ and a string $z \in \Sigma^*$ such that $xz \in L \iff yz \notin L$. The variant query, $\mathrm{QueryVariantDB}$, searches all possible suffixes in the database. Because we assume the database contains sufficient information, such a z will be discovered. Since there are only two underlying states, and such a partition with z makes two different equivalence classes, one will correspond to one state and the other to the other.

Theorem *Correctness*: The algorithm terminates with a DFA that is isomorphic to the minimal DFA accepting L.

Proof We need to show that (1) every state of the DFA is found, and (2) every transition is found.

(1) States. Assuming every state is reachable, we can use induction on the structure of the DFA.

- **Base case:** The initial state is discovered because the algorithm starts with the empty prefix ε in C_0.
- **Inductive step:** Assume a state q whose representative prefix is p, has been discovered. Let s be a successor state of q through symbol $a \in \Sigma, \delta(q, a) = s$. After each splitting phase, the algorithm expands the frontier by adding all missing transitions (Lines 14, 15 of algorithm 9). Thus, after q is found, the string $p \cdot a$ is either already or will be added to some equivalence class C'.

  There are three cases for C':

  1. C' contains only prefixes leading to s. By the Myhill-Nerode theorem, there are no distinguishing suffixes, so $\mathrm{SplitClassVariant}$ returns $\emptyset$ and C' is finalized as the class for state s.
  2. C' contains prefixes from more than two different states. Then, by Myhill-Nerode, distinguishing suffixes exist, and they will be found by $\mathrm{SplitClassVariant}$ and so C' will be split until it reaches the following case.
  3. C' contains $p \cdot a$ and prefixes from exactly two states (so one of them is s). By the lemma, $\mathrm{SplitClassVariant}$ splits them into two classes, one of which corresponds to s.

  In every case, the state s is discovered.

(2) Transitions After every split, all missing transitions are found and added (as one-symbol extensions of a representative of the class). This also happens after the last split so no transitions are missed.

Since the states were partitioned with the Myhill-Nerode theorem, the resulting DFA is minimal.

Since splits create one new equivalence class each time and there are finitely many classes, the algorithm terminates.

7.3. Number of States Discovered by Algorithms / Correctness

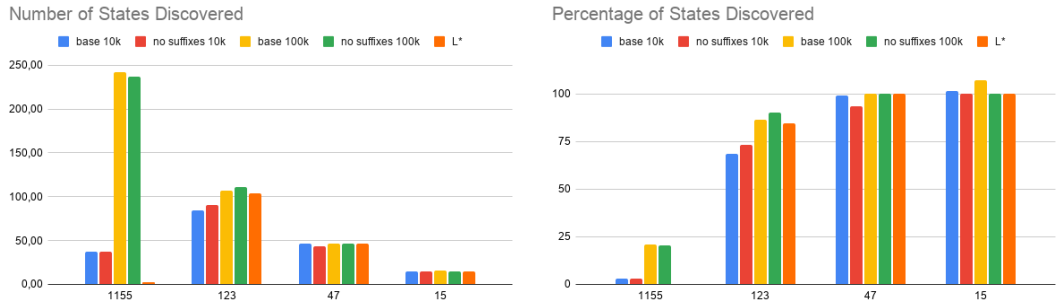


Figure 7.2: Number of states discovered by different methods and datasets (left) and also as a percentage of total states (right).

Among the experiments run, I also recorded the number of states the algorithms output. This gives an indicator of the correctness and can reveal how well the models fare against different dataset types. Figure 7.2 depicts the average number of states discovered by each binary multiples algorithm. From this, we can draw some interesting conclusions.

First of all, we can see that the base algorithm overshoots in multiples of 15. That means it accidentally detects more states than necessary, and some of the states it outputs could be merged. Since the no suffixes variant never has such behavior, this is likely related to the populate query. There may be a bug in the implementation that makes certain prefixes pop up in the wrong class.

Additionally, in almost all cases, the “no suffixes” variant discovers more states than the base variant on the same dataset size. The most likely explanation for this is that the pumped suffix of the regex is more dependent upon a “good” prefix set in a state. This is because the prefix set determines the suffixes in the regex, and show determines how much of the dataset will be explored. More possible suffixes mean a higher chance of finding a distinguishing suffix in the database, and I have already demonstrated the example in section 7.1, in which the pumped suffixes fail to detect a distinguishing suffix. As such, the no suffixes method can more easily find distinguishing suffixes, as its suffix is not constrained, but at a higher cost of time, as we saw. Better calibration of the population parameter can potentially improve the accuracy of the base method.

Moreover, we can see that with larger dataset sizes ($\geq$ 100k) and at more complicated machines with more states, our methods perform better than L*. In smaller machines, the performance is the same. Note that in multiples of 1155, L* gives up and only detects two states.

An interesting conclusion is that these methods always perform better with increasing dataset size, even seeing a drastic improvement when going from 10k traces to 100k in multiples of 1155. This further highlights their suitability for large datasets, besides the RAM issues that passive learning methods face. On the other hand, as expected, the more complicated the model, the harder it is to find a correct automaton for it.

Figure 7.3 depicts the percentage of discovered states for the “k-th symbol is 0” datasets. The behavior differs from the binary multiples.

The base variant tends to overshoot quite a bit (the actual values were cropped at 125 to better illustrate the graph, but it reaches about 140%). However, in 10th, it appears to converge towards 100% with increasing dataset size.

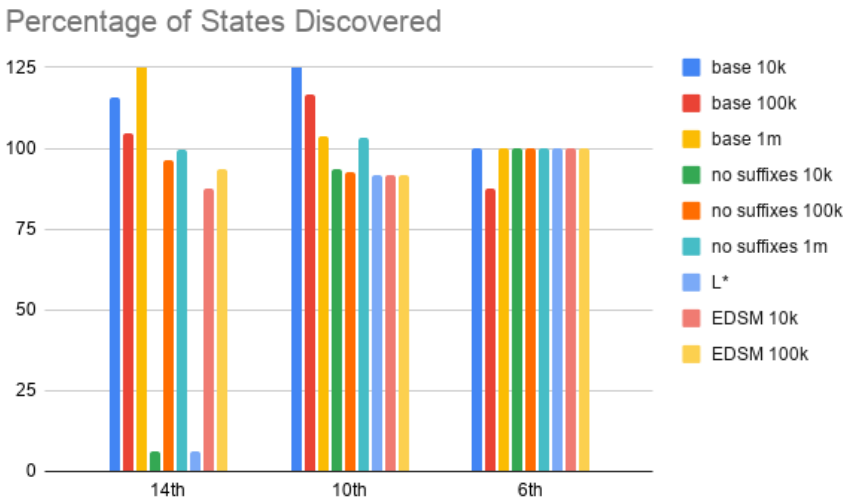


Figure 7.3: Percentage of states discovered in the “k-th bit is 0” datasets by different algorithms.

Convergence towards 100% with larger dataset sizes is also observed in the no suffixes variant in 14th and 10th (in 6th, all are already at 100%).

We can also see that in 14th, the no suffixes with 10k traces and L* fail to reach close to the target DFA.

Additionally, across the same dataset sizes, the “no suffixes” variant consistently outperforms EDSM by a small margin. EDSM always finds the same number of states and the same lacking automaton. However, the no suffixes variant sometimes identifies the correct automaton, though sometimes it also finds the same as EDSM. Additionally, on the 1m dataset on which EDSM fails, “no suffixes” produces an even better automaton on average, and scaling further seems to be in favor of both variants. The base variant also finds more states than EDSM, but it overshoots, as already discussed.

7.3.1. Deviations Explained

The reason the “no suffixes” variant fails to achieve 100% every time, despite the correctness proof, is that the proof assumes the traces available in the dataset are infinite. In the experiments, however, they were limited to 100k or 1 million, so the search for distinguishing suffixes may have failed because the necessary traces were not present in the database. This is supported by the fact that in all cases, increasing the dataset size produces more correct results.

Besides, L* also has a correctness proof and still fails or undershoots in some cases.

7.4. Evaluation on the Abbadingo Dataset

In figure 7.4, we can see two plots. Both depict the percentage of states discovered for the abbadingo datasets as a function of the target DFA depth, except for 3 datasets, for which the algorithm overshot. In the top one, the number of traces in each dataset is annotated, and in the bottom one, the number of states in the target DFA.

We can notice some interesting trends that give intuition about under what circumstances the algorithm performs well or not.

First, we can clearly see that for the same target DFA depth, more traces in the dataset lead to more discovered states. This is expected because it’s more likely that for a given set of prefixes, the distinguishing suffix exists in the database.

Second, we see that, in general, the deeper a DFA is, the worse the algorithm’s performance is. This happens because, in order for the deepset states to be discovered, the predecessors need to be discovered first. Since the query uses a sample of prefixes leading to a predecessor state, it has a chance

Figure 7.4: Average percentage of states discovered vs. target DFA depth, annotated with dataset size (top), and total number of states in the target DFA (bottom)

to find the distinguishing suffix. In a long chain of states, this probability accumulates, making it less likely to find the last one, because that depends on having found all previous ones.

Third, contrary to the previous observation, we can notice an increase in performance when the target DFA depth is 12. The reason for that is revealed by the target DFA states. The datasets with depth 10 have approximately 60 total states, giving a rough estimate of 6 states per depth level. In depth 12, we have about 130 total states, which gives a rough estimate of about 10 states per depth level. This means the depth 12 DFAs are more likely to be better connected and to have multiple paths even to deeper states than the depth 10 DFAs. This means it's more likely for states to be discovered in general. The conclusion is that the more connected a DFA is, the better the implementation of sampling the states works.

7.5. Summary

This chapter outlined both theoretical and empirical insights into the correctness of the split-explore loop. The simple variant with the pumped regex has a blind spot that makes a correctness proof much harder for it. On the other hand, the "no suffixes" variant that examines all possible suffixes has a clean proof under the assumption that all states in the target DFA are reachable and that the database contains sufficient information regarding them. The key step of that algorithm that enables the proof is the addition of missing transitions after each split.

In the experiments, we see both variants outperform L* on harder datasets and perform about the same as EDSM at the same dataset sizes. However, because they can scale much more than EDSM, and because scaling produces better models, eventually the 1 million models tend to be better than what EDSM can produce. While overshooting occurs sometimes in the split-explore loop, the output DFA appears to converge to the correct one with larger dataset sizes.

An evaluation on the Abbadingo dataset, which is more diverse and potentially more realistic, provides additional evidence for these conclusions. DFA accuracy improves with the number of traces in the dataset, which is particularly effective when combined with scaling. Deeper DFAs pose a greater challenge because discovering deep states depends on finding their predecessors. High connectivity mitigates this effect, and this kind of algorithm performs better in highly connected DFAs.

8

Conclusion

In conclusion, model learning algorithms using databases offer a unique solution to the problem of massive datasets and logs produced by a good deal of real-world software. This thesis explores whether regex queries in databases can be utilized to build automata learning algorithms that can scale beyond RAM constraints, limiting typical passive learning. I started with a smaller algorithm implemented using PostgreSQL's `regexp_matches`, then developed two algorithm variants that autonomously discover more states. I provided a proof of correctness and compared the methods with both active and passive learning algorithms across various datasets, both in their sizes and in the complexity of their automata.

This chapter re-examines the research questions, gives answers to them, and provides a discussion on possible interesting future work.

8.1. Research Questions and Answers

The research questions answered in this thesis were the following in order of appearance.

RQ1 How does the initial algorithm compare with typical passive learning methods in terms of runtime and scalability?

In Chapter 3, the initial algorithm is tested for this exact question.

The most important advantage of this kind of algorithm over passive learning methods is that it can scale to large dataset sizes. As shown by EDSM, passive learning methods require the entire dataset to be in RAM, which at 1.5 million traces becomes infeasible, running out of RAM and crashing, while the initial version of the algorithm could correctly complete the task.

Additionally, the log-log scale plot demonstrated linear growth in execution time with dataset size. The initial algorithm was faster than EDSM in all cases, despite it being written in Python versus the optimized C++ implementation in FlexFringe. It also did not have any memory-related failures, meaning it provides scalability for real-world tasks where logs can reach millions or billions of traces.

The optimization of skipping the first large regex provided a huge speedup of 2.4, reducing the execution time from 23 to about 10 seconds in the 1.5 million datasets.

RQ2 With active learning methods in terms of query efficiency?

The initial algorithm proves to use orders of magnitude fewer queries when compared to typical active learning methods like L*. It makes exactly *two queries per state* of the target DFA. One to split each state and one to confirm the state cannot be further split, compared to the hundreds or thousands of queries L* needs.

The reason behind this is that regex queries can condense thousands of individual membership queries into a single regex. In applications where queries are costly or time-consuming, this method can offer

real advantages.

RQ3 How can the algorithm be expanded with the aim of discovering more states autonomously, without specific human input or guidance?

Chapter 4 covers this in full detail.

The initial version of the algorithm depended upon an initial prefix set, which had to be representative of each state in the target automaton. The naive enumeration of all substrings up to a length would generate an exponential set in cases where some states require long inputs.

To overcome this, or the need for a carefully designed prefix set, I created some methods to expand the prefixes used based on a decision tree-like structure, recording all splits. `add-prefixes` takes a number of prefixes to be added and detects the appropriate candidate class for each by testing it with each distinguishing suffix it would have been tested with, had it been present since the beginning. On the other hand, the PopulateClass query does the inverse. Given a class, it produces a regex through the path of the class in the decision tree and uses the regex in the database to produce prefixes of that class.

These two methods enable re-expressing the algorithm in a two-phase loop with a split phase and a discovery phase. To autonomously detect more states, missing transitions to other classes can be added after a split, and before examining a class for a split, a number of prefixes of it can be generated to widen the search for possible splits. The extended algorithm correctly discovered the automaton for languages where the 6th and 16th bits are set to 0, starting only with the alphabet, 0, 1. Additionally, a real-world application on a small HTTP server correctly produced a state machine of the server's request handling.

RQ4 How does the expanded algorithm compare in the same ways?

Chapter 6 demonstrates the relevant results.

Two variants of the extended algorithm were tested. The one named *base* used the pumped suffix in the regex, which reduces the search space for distinguishing suffixes. The one named *no suffixes*, which uses the variant query examining all possible suffixes and does not benefit from the populate step.

In all cases tested, binary multiples of 15, 47, 123, and 1155, of sizes both 10k and 100k, the no suffixes variant was faster and discovered more correct states. This is due to the regex searching all possible suffixes, and so it doesn't miss possible distinguishing suffixes.

The variants of the expanded algorithm appear to be slower than simple passive learning due to their more complex queries, however the still scale to much bigger datasets. They use orders of magnitude fewer queries than L*. In addition, they produce more correct automata compared to L* and about the same in terms of correctness with EDSM. In the experiment of 1155, L* would return a machine with only two states instead of the correct one with $\sim$1000. In contrast, the two variants discovered machines with hundreds of states and had a large improvement when scaling from 10k to 100k samples, something which is not observed in passive learning methods, which, after some point, run out of RAM.

RQ5 Can we give a correctness proof for the algorithm?

Correctness is analyzed in detail in Chapter 7.

We can give a correctness proof for the simpler, "no suffixes" variant of the algorithm, which searches for all suffixes. A correctness proof for the more specific algorithm is more difficult, and that version might require some adjustments, as will be discussed in future work.

The experiments on the number of states support this, too. Additionally, experiments on the Abbadingo datasets indicate under which conditions of the target DFA the algorithm performs best.

8.2. Future Work

There are quite a number of directions that research into this style of algorithm can still look into.

8.2.1. Adding new prefixes

First, alternative ways of adding new prefixes can be examined. Instead of adding missing transitions after splitting and populating before splitting, one could develop criteria to make more targeted and informed additions instead. It could also be possible to utilize information contained in the distinguishing suffixes and them, or parts of them, as, or derive from them, new prefixes in a reasonable way. Moreover, another approach could be to keep each class to some number of population by using the populate method to add a number of prefixes when it loses some to a split, and when it's generated with really few prefixes.

8.2.2. A fully in-the-database algorithm

Since both the splitting operations and the addition of new prefixes happen inside the database, what's left in the CPU is just keeping track of the loop and the sets. The populate operation finds which prefixes in the database belong to each class. Perhaps it is possible to replace the class samples stored in the CPU with a population query and move all processing to the database, yielding an algorithm that entirely runs in the database. This has several advantages. It might be faster since no time will be wasted in communicating from and to the disk, and database optimizations might also lend a hand. Furthermore, there would be no danger of missing splits because the entirety of a class in the database would be used for each split with a populate query.

8.2.3. Nondeterminism

The problem of automata learning has been known to be NP-Complete [4]. As such, nondeterminism is a crucial aspect of this problem that needs to be dealt with. In this algorithm, nondeterminism appears when the database does not hold information for a prefix followed by the distinguishing suffix (so we don't know in which of the split classes it should be added) or when adding a prefix through the decision tree, and one of the suffixes does not exist in the database. In both cases, the implementation treats the nonexistence as a negative response to the query and proceeds with that, as in both cases, there is no evidence that the prefix should be split, and there is no evidence that the query should move into the next class. However, when this happens, perhaps it would be more correct to move the prefix in both classes, and for the addPrefixes method to return a set of possible classes the prefix might belong to. Perhaps it would be possible then to maintain a conflict graph based on the previous queries, which would help resolve some of these nondeterminisms. Then hopefully a small amount of them is left, and it can be resolved by checking all possibilities.

8.2.4. SQL/Query Optimizations

Finding ways to improve query speeds is critical for this algorithm. As shown in the initial results, most of the runtime of the algorithm is dominated by the queries. Moreover, the experiments demonstrate that for one variant, the populate query is responsible for most of the time taken, and for the other, the regex query. Perhaps the `(.*)$` regex query can be optimized further than the implementation PostgreSQL provides through the use of a trie or a prefix tree.

8.2.5. Conflict Graph

A conflict graph could be utilized to store information about which prefixes conflict and cannot be in the same state. This could reduce the number of times a database query is made, thus improving the performance of the algorithm.

It could also provide a solution for the blind spot presented in Chapter 7.

8.2.6. Other Automata Types

Variants of the algorithm could be developed for learning different kinds of automata, such as Mealy/-Moore machines, symbolic automata, bidirectional automata, etc.

References

[1] Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. “Benchmarking Combinations of Learning and Testing Algorithms for Automata Learning”. In: *Form. Asp. Comput.* 36.1 (Mar. 2024). issn: 0934-5043. doi: 10.1145/3605360. url: https://doi.org/10.1145/3605360.

[2] Dana Angluin. “Learning regular sets from queries and counterexamples”. In: *Information and computation* 75.2 (1987), pp. 87–106.

[3] George Argyros. “Another Brick O the Wall : Deconstructing Web Application Firewalls Using Automata Learning”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016. url: https://api.semanticscholar.org/CorpusID:113400134.

[4] E Mark Gold. “Complexity of automaton identification from given data”. In: *Information and Control* 37.3 (1978), pp. 302–320. issn: 0019-9958. doi: https://doi.org/10.1016/S0019-9958(78)90562-4. url: https://www.sciencedirect.com/science/article/pii/S0019995878905624.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd. Addison-Wesley, 2006. isbn: 978-0321455369.

[6] Kenneth L. Ingham et al. “Learning DFA representations of HTTP for protecting web applications”. In: *Comput. Netw.* 51.5 (Apr. 2007), pp. 1239–1255. issn: 1389-1286. doi: 10.1016/j.comnet.2006.09.016. url: https://doi.org/10.1016/j.comnet.2006.09.016.

[7] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. “Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm”. In: *Grammatical Inference*. Ed. by Vasant Honavar and Giora Slutzki. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–12. isbn: 978-3-540-68707-8.

[8] Edi Muškardin et al. “AALpy: An Active Automata Learning Library”. In: *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings*. Gold Coast, QLD, Australia: Springer-Verlag, 2021, pp. 67–73. isbn: 978-3-030-88884-8. doi: 10.1007/978-3-030-88885-5_5. url: https://doi.org/10.1007/978-3-030-88885-5_5.

[9] Jose Oncina and Pedro García. “Inferring regular languages in polynomial update time”. In: *World Scientific* (Jan. 1992). doi: 10.1142/9789812797902_0004.

[10] Doron Peled. “Reverse Engineering Through Automata Learning”. In: *Formal Methods in Outer Space: Essays Dedicated to Klaus Havelund on the Occasion of His 65th Birthday*. Ed. by Ezio Bartocci, Yliès Falcone, and Martin Leucker. Cham: Springer International Publishing, 2021, pp. 182–192. isbn: 978-3-030-87348-6. doi: 10.1007/978-3-030-87348-6_11. url: https://doi.org/10.1007/978-3-030-87348-6_11.

[11] Harald Raffelt and Bernhard Steffen. “LearnLib: A Library for Automata Learning and Experimentation”. In: *Fundamental Approaches to Software Engineering*. Ed. by Luciano Baresi and Reiko Heckel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 377–380. isbn: 978-3-540-33094-3.

[12] R.L. Rivest and R.E. Schapire. “Inference of Finite Automata Using Homing Sequences”. In: *Information and Computation* 103.2 (1993), pp. 299–347. issn: 0890-5401. doi: https://doi.org/10.1006/inco.1993.1021. url: https://www.sciencedirect.com/science/article/pii/S0890540183710217.

[13] Michael Sipser. *Introduction to the Theory of Computation*. 3rd. Cengage Learning, 2013. isbn: 978-1133187790.

[14] Sicco Verwer and Christian Hammerschmidt. “FlexFringe: Modeling Software Behavior by Learning Probabilistic Automata”. In: *Logical Methods in Computer Science* Volume 21, Issue 3, 31 (Sept. 2025). issn: 1860-5974. doi: 10.46298/lmcs-21(3:31)2025. url: https://lmcs.episciences.org/9295.

[15] Hielke Walinga, Robert Baumgartner, and Sicco Verwer. *Database-assisted automata learning*. 2024. arXiv: 2406.07208 [cs.FL]. url: https://arxiv.org/abs/2406.07208.

A

Implementations of Queries in SQL

A.1. QueryDB - Query that splits a class

The SQL implementation of QueryDB(C, label) is presented below. Since it is used within Python, the %s represents a variable part of the query, and its first occurrence will be replaced by the custom regex of the equivalence class under examination, while its second with a boolean variable representing whether the class is accepting or not (used to filter only for counterexamples).

```
WITH matched AS (
    SELECT (m)[1] AS prefix, (m)[2] AS suffix
    FROM traces, regexp_matches(trace, %s) AS m
    WHERE accepting = NOT %s
),
suffixes AS ( -- speeds up the subsequent ORDER BY
    SELECT DISTINCT suffix
    FROM matched
),
distinguishing_suffix AS (
    SELECT suffix
    FROM suffixes
    ORDER BY length(suffix) -- shortest counterexample
    LIMIT 1
)
SELECT prefix
FROM matched
WHERE suffix = (SELECT suffix FROM distinguishing_suffix)
```

A.2. Populate Query

The implementation is presented below. The first %s corresponds to S^+, and the second to S^-. The following two are part of the set of prefixes of the target equivalence class (small optimization to speed up the query). The last %s corresponds to the parameter n of how many examples to return.

```
WITH pos_prefixes AS (
    SELECT
        LEFT(trace, LENGTH(trace) - LENGTH(s.suffix)) AS prefix
    FROM traces
    CROSS JOIN UNNEST(%s) AS s(suffix)
    WHERE trace LIKE '%%' || s.suffix
    GROUP BY 1
    HAVING BOOL_AND(accepting)
),
neg_prefixes AS (
    SELECT
        LEFT(trace, LENGTH(trace) - LENGTH(s.suffix)) AS prefix
    FROM traces
    CROSS JOIN UNNEST(%s) AS s(suffix)
    WHERE trace LIKE '%%' || s.suffix
```

```
    GROUP BY 1
    HAVING NOT BOOL_OR(accepting)
)
SELECT prefix
FROM (
    SELECT prefix FROM pos_prefixes
    WHERE NOT (prefix = ANY(%s)) -- not already in class
    INTERSECT
    SELECT prefix FROM neg_prefixes
    WHERE NOT (prefix = ANY(%s)) -- not already in class
)
LIMIT %s;
```

B

Dataset Generation Code

By leveraging Python *generators*, we can write one-liners that generate the desired datasets. However, the full version of the code still includes some boilerplate for connecting to PostgreSQL through the `psycopg2` library and creating a table.

The code I created employs several tricks that the reader might find interesting:

m binary multiples of n

```
    cur.executemany("INSERT␣INTO␣traces␣(trace,␣accepting)␣VALUES␣(%s,␣%s)",
                    ((f"{i:b}", i % n == 0) for i in range(m)))
    cur.execute("INSERT␣INTO␣traces␣(trace,␣accepting)␣VALUES␣(%s,␣%s)", ('', True))
```

k-th character is c

```
    def kth(k, c, size):
        for i in range(size):
            s = f"{i:b}"
            yield s, k - 1 < len(s) and s[k - 1] == c

    cur.executemany("INSERT␣INTO␣traces␣(trace,␣accepting)␣VALUES␣(%s,␣%s)", kth(k, '0', n))
    cur.execute("INSERT␣INTO␣traces␣(trace,␣accepting)␣VALUES␣(%s,␣%s)", ('', False))
```