Database-assisted state machine learning

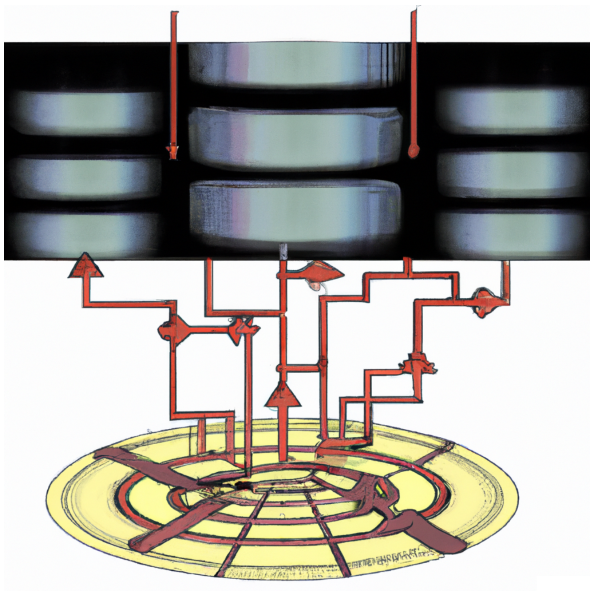
Hielke Walinga
student number: 4373561

Presented for the fulfillment of the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE



Algorithms group
Technische Universiteit Delft
The Netherlands
30 May, 2024, 14.00

Thesis committee:

Prof. dr. ir. Sicco Verwer	Algorithms group	TU Delft	Thesis advisor
Dr. Rihan Hai	Web Information Systems group	TU Delft	Committee member
Ir. Robert Baumgartner	Algorithms group	TU Delft	Daily co-supervisor

Abstract

The behavior of software systems can be modeled as state machines by looking at the log data from these systems. Conventional algorithms, such as L^*, however, require too much memory to process log data when it gets too large. These algorithms must first load all available data into memory, which is often way too much.

The approach investigated to deal with this large amount of data is to load the data instead in an on-disk database. The thesis researches the viability of this approach and proposes an algorithm for it.

When log data is present in a database, a state machine can be constructed iteratively by gradually querying more and more data from the database. Each time more data is gathered, partial data can be used to construct a hypothesis for the state machine in question. Such an approach is thus a combination of an *active learning* part that queries more data and an *passive learning* part that constructs the state machine from partial data.

Database technologies can contribute to what such an algorithm should look like. Given the vast landscape of different database technologies, some can shape what kinds of queries can be more efficiently executed. This should be taken into account when constructing an algorithm that queries data from a database to construct a state machine.

This thesis starts with an overview of the existing state machine learning algorithms and relevant database technologies. Then a new algorithm is proposed and it is tested on different datasets and compared with existing algorithms.

Preface

After almost 10 years, this thesis concludes my academic career at the TU Delft. It was long enough. After a setdown during the Corona pandemic, I was looking to go back to the basics. I was looking for a project that was closer to the core of computer science.

This project fitted perfectly with what I had in mind. On top of that, in the meantime I was able to learn a programming language that is the basis of many of our useful technologies.

As this thesis is an investigation in a fairly new research direction, I mostly tried to write it as an introduction to all its moving components. This thesis thus could act as a good starting point for anybody starting a similar project continuing my work. This thesis is thus not only a scientific report but also an engineering manual.

Preliminary results from the thesis have also been submitted to the LearnAut24 workshop at ICALP in Tallinn (Estonia), 7th of July 2024[1]. If accepted, I will also have the opportunity to present this work to experts in the field at this workshop. The paper is included in this thesis after the introduction.

For the people that are reading this because they are pursuing this research direction, I wish them good luck. You are free to reach out to me if you have any questions. And for those that are reading this because they have to grade me, I wish you fun reading.

"Learning is all about forgetting."

Colin de la Higuera - *Grammatical Inference (2010)*

Hielke Walinga
hielkewalinga@gmail.com
May 2024

[1] https://learnaut24.github.io

Acknowledgements

Hereby I express some gratitudes.

I want to thank my parents for their love, universal life skills and lessons, and the gift of perseverance.

I want to thank my girlfriend for her love, support, and having fun together.

I want to thank my friends for sharing the burden of completing a university degree and their encouragement of completing mine.

I want to thank Robert, who has helped me through all my daily struggles with his thoroughness, in-depth knowledge, and his good vibes.

I want to thank Sicco for the opportunity to take upon the project and for his never-ending stream of good ideas and his enthusiasm which has been a true inspiration.

I want to thank Rihan for being on the thesis committee and for her quick communication.

The graphic art on the title page is generate with DALL-E 2[2] with the terms "art deco digital rendering rendering of a pathway diagram with database, keep a white background".

Figure 1: A bunny with a backpack. (Requested by my girlfriend.)
Image courtesy to https://www.pinterest.com

[2]https://labs.openai.com/

Contents

1 Introduction **7**
1.1 Relevance . . . 7
1.2 Context . . . 8
1.3 Research gap . . . 8
1.4 Research questions . . . 9
1.5 Methodology . . . 9
1.6 Contributions . . . 10

Publication for LearnAut24 **11**

2 Background **23**
2.1 State machines . . . 23
2.2 Inference . . . 24
2.2.1 Active learning . . . 24
MAT . . . 24
L^* algorithm . . . 25
2.2.2 Passive learning . . . 27
State merging . . . 27
Blue-fringe algorithm . . . 27
Example . . . 28
2.3 Databases . . . 31
2.3.1 Indexes . . . 31
B-tree . . . 31
GIN . . . 31
GiST . . . 31
SP-GiST . . . 31
2.3.2 Text search . . . 32

Full-text search and trigramming . . . 32
Operator families . . . 32
Collations . . . 32
2.3.3 Prefix query . . . 33
Implementation . . . 33
B-tree vs SP-GiST . . . 33
Radix tries and the ART index . . . 33
2.3.4 Miscellaneous . . . 33
Including columns inside an index . . . 33
The COPY protocol of PostGreSQL . . . 33
2.4 State of the art . . . 34
2.4.1 FlexFringe . . . 34
2.4.2 More research on L^* . . . 34
$L^\#$ Algorithm . . . 35
iMAT . . . 36

3 Methods **38**
3.1 Database queries . . . 38
3.1.1 Prefix query . . . 38
3.1.2 Distinguish query . . . 39
3.1.3 Regex query . . . 39
Regex construction . . . 39
3.2 Algorithm: DAALder algorithm . . . 40
3.2.1 Motivation . . . 40
3.2.2 DAALder algorithm outline . . . 41
3.2.3 Differences with $L^\#$ . . . 41
Blue node completion . . . 41
A FlexFringe limitation . . . 43
Other differences . . . 43
3.2.4 Exploration during merging . . . 43
Discovering the unidentified portions . . . 43
Staying close to passive learning . . . 43
Adding exploration . . . 44
Example . . . 44
With distinguishing queries . . . 44
Equivalence oracles . . . 44
Regular expression . . . 44
Distinguishing queries . . . 45
Randomized queries . . . 46
3.3 Experiments . . . 46
3.3.1 Test data . . . 46
Stamina . . . 46
FSM-learning generator . . . 46
3.4 Implementation . . . 47

4 Results and discussion **48**
4.1 Failures . . . 48
Stamina . . . 48
Regex . . . 48
One data source . . . 48
4.2 Database performance . . . 49
4.2.1 Index types performance . . . 49
Discussion . . . 51
4.3 Performance of algorithms . . . 51
4.3.1 Comparing algorithms . . . 51
Results . . . 51
Discussion . . . 54
Solvability point . . . 54
Runtime comparison . . . 54
A characteristic set . . . 54
Memory usage . . . 54
Effect of k . . . 55
Accuracy comparison . . . 55
4.4 Miscellaneous . . . 55
4.4.1 Data loading . . . 55
4.4.2 Does sampling work? . . . 56
4.5 When to pick DAALder . . . 56

5 Future work **57**
5.1 Braoder comparison with different algorithms and datasets . . . 57
5.1.1 Different algorithms . . . 57
5.1.2 Different datasets . . . 57
5.2 Implementation improvements . . . 58
5.2.1 Improved regex building . . . 58
Different regex constructing algorithms . . . 58
Selecting the most important parts . . . 58
5.2.2 Being able to add traces to a partial mealy machine . . . 58
5.3 Extensions . . . 59
5.3.1 Investigation into more query types . . . 59
5.3.2 Investigation into the search algorithm . . . 59
5.3.3 Other forms of the state machine . . . 59
Using the guards of FlexFringe . . . 59
Adding statistics to the database . . . 59
5.3.4 Incorporating data integration techniques . . . 59
5.4 Different implementations . . . 60
5.4.1 Using another database more tailored towards the problem . . . 60
5.4.2 RIIR . . . 60
5.4.3 Circumventing implementation details altogether using Datalog . . . 60

6 Conclusion **62**
6.1 Research questions . . . 62
6.1.1 RQ1: What kind of queries can we ask a database to learn state machines? . . 62
6.1.2 RQ2: What kind of algorithm can we use to learn a state machine from a database? 62
6.1.3 RQ3: How can we measure the performance of such an algorithm and how does this algorithm perform . . . 63
6.2 About learning . . . 63

Bibliography **64**

CHAPTER 1

Introduction

1.1 Relevance

In software engineering and cybersecurity, state machines can serve as models for a software system. How the software system can behave is concisely captured within such a state machine. These state machines provide a model to reason about the software systems. This model can help to either find bugs or monitor any unexpected behavior of the software system as a means to protect against malicious agents in a cybersecurity setting.

A state machine, or specifically a deterministic finite-state machine (DFA), is a mathematical model consisting of a finite number of states with transitions between them. This machine can be applied to model what output to expect from a system given a certain input. Another application for a state machine is to mathematically describe a formal language. For an arbitrary string, the state machine can be used to tell if that string is part of the language this state machine describes.

After the state machine is constructed for a software system, it proves to be versatile in its usage. With the state machine, we can see where unexpected transitions are different from the specifications of the system. This could indicate that there is a bug in your system. We can also test using a state machine if previous behaviors of the system are not expected and thus can indicate something malicious. Thus, these state machines can be used to test software systems [1], to look for cyber attacks [2, 3], to check protocols [4, 5], to reverse engineer protocols [6, 7], or to verify the correctness of implementations, such as web protocols [8], X11 programs [9], TCP [10], or SSH [11].

Learning such state machines often involves analyzing the output of software systems. For software systems, there is often log data to look at, but the state machine can also be learned by actively providing data to the software system to see how it reacts. The amount of data needed to learn a state machine can be significantly large. So much so that it often cannot be loaded into memory. However, this data consists of lots of similar data. It also often contains very repetitive structures in the data. Finding ways to extract useful information quickly will greatly improve how fast we can learn state machines from large amounts of data.

This thesis is focused on these ways of learning state machines from very large datasets. It explores database-based strategies to solve the problems large sets of data entail, by constructing clever queries to the database to use while learning the state machine.

1.2 Context

Algorithms to infer DFA's can be divided into two categories. One is *active learning*, online algorithms that construct the state machine by actively providing clever inputs to the system and, during that process, learn the state machine. The other is *passive learning*, offline algorithms that process a large amount of inputs and outputs, and infer from that data the underlying states. Active learning has the disadvantage that the system has to be present to be interrogated during learning. Passive learning, on the other hand, will often require a substantial amount of data to cover all parts of the system since you do not have access to the system to explore its unknown territory. Thus for passive learning, all the initial data must already provide all the information to construct the corresponding state machine.

In active learning, you learn the state machine of your system by querying the system you are interested in. The system here is often a software system and is called the system under learn (SUL), or sometimes the system under test (SUT). Instead of randomly asking queries to the SUL, the goal is to only ask specific queries for the information you do not know already, and thereby limiting the number of queries you ask. This can be used only if a system is available for querying when learning the model.

Passive learning algorithms utilize a fixed set of traces available to the learner to derive the state machine from. Due to the fact that learning DFAs from trace data is NP-hard [12], passive learning algorithms necessitate some kind of heuristic method to be performant. It is thus a goal of passive learning to identify from a given set of traces, and in accordance with Occam's razor, the smallest state machine that best explains that set of traces. The inherent challenge here is to find a significantly compact state machine that explains the set of traces well enough without over-fitting to the given dataset. It is not required to find the smallest state machine to explain the data, but size can be seen as a performance criterion [13].

1.3 Research gap

Instead of using passive learning algorithms on passive data, we can also use active learning algorithms that ask questions on the passive data. Here, the passive data acts like the system under learn without the actual system present, and the passive data does not have to be processed completely. This can be seen as active learning on passive data.

This allows us to comfortably learn from very large data, as this data can be saved in the database without utilizing an enormous amount of RAM, and for which its query performance scales reasonably well. Saving to the database and indexing can be seen as a form of pre-filtering on the data. With the use of clever indexing structures and modified active learning algorithms, we can still learn state machines from this data in a reasonable amount of time.

The problem this entails is that active learning algorithms often assume that all queries can be answered fully. However, now there is only partial data available. The algorithms we use, therefore, should be able to deal with this missing data. In essence, the database provides here what the iMAT teacher would be (see section 25) [14].

This approach also brings more possibilities on what kind of queries we can ask the database. If the database is cleverly designed, we can ask more advanced queries directly on the database that can aid the algorithm in learning the state machine more efficiently.

1.4 Research questions

Our main research question is if this way of learning state machines is possible. This research question thus boils down to the following:

How can we learn state machines from large sets of trace data using a database?

This question has subquestions. First, we want to look into what kind of queries can we efficiently ask this database, thus:

RQ1: What kind of queries can we ask a database to learn state machines?

Then, how can we learn from such a database? We look for how to construct an algorithm that can learn with these kinds of queries, thus:

RQ2: What kind of algorithm can we use to learn a state machine from a database?

Lastly, we want to check the performance of this algorithm. This means we first have to figure out how to do this and then measure the performance on some datasets.

RQ3: How can we measure the performance of such an algorithm, and how do these proposed algorithms perform?

1.5 Methodology

First, the current state of state machine learning is given with a description of important algorithms. Then, an overview of the design of the database and what queries it can answer is given. Second, an outline of the algorithm that can make use of these queries is given. Subsequently, this algorithm with the database is tested on various test datasets.

1.6 Contributions

This thesis gives an extensive description of the moving parts this project entails. This includes a literature review of state machine learning in general, including a focus on specific research related to the approach to the problem of this thesis. This also comes with a section on the technologically important details of databases, as this is important for our research.

This project also contributes a component in the FlexFringe active learning part framework that can load traces in a database and learn from it. The component is made inside the active learning framework, and the SUL class is wired to a database. This will also make part of the implementation reusable and extensible for further research.

What now follows first is a scientific contribution in the form of an extended abstract submitted and accepted to the LearnAut24 workshop at ICALP. This paper contains a brief summary and some preliminary results. More extensive results can be found in chapter 4 on page 48.

Database-assisted automata learning

Hielke Walinga H.Walinga@student.tudelft.nl
Robert Baumgartner R.Baumgartner-1@tudelft.nl
Sicco Verwer S.E.Verwer@tudelft.nl
Software Technology Department, EEMCS, Delft University of Technology, Delft, the Netherlands

Abstract

This paper presents DAALder (Database-Assisted Automata Learning, with Dutch suffix from *leerder*), a new algorithm for learning state machines, or automata, specifically deterministic finite-state automata (DFA). When learning state machines from log data originating from software systems, the large amount of log data can pose a challenge. Conventional state merging algorithms cannot efficiently deal with this, as they require a large amount of memory. To solve this, we utilized database technologies to efficiently query a big trace dataset and construct a state machine from it, as databases allow to save large amounts of data on disk while still being able to query it efficiently. Building on research in both active learning and passive learning, the proposed algorithm is a combination of the two. It can quickly find a characteristic set of traces from a database using heuristics from a state merging algorithm. Experiments show that our algorithm has similar performance to conventional state merging algorithms on large datasets, but requires far less memory.

Keywords: Active/Passive state machine learning, Incomplete Minimally Adequate Teacher

1. Introduction

Automata can be helpful models for analyzing software systems. For example, state machines have helped to analyze software for web protocols (Bertolino et al., 2009), X11 programs (Ammons et al., 2002), and SSH (Fiterău-Broştean et al., 2017). They have also helped to test software systems (Hagerer et al., 2001), to check software protocols (Cho et al., 2011; Comparetti et al., 2009), or for reverse engineering (Antunes et al., 2011; Cui et al., 2007).

Finding a corresponding state machine for a software system is a grammatical inference problem (de la Higuera, 2010). Techniques for inferring the state machine can be divided into two categories. One is *active learning*, algorithms that construct the state machine by providing clever inputs to the system and, during that process, learn the state machine (Angluin, 1987). The other is *passive learning*, algorithms that process a large amount of inputs and outputs, and infer from that data the underlying states (Cook and Wolf, 1998).

A problem for *passive learning* is that all input data has to be present at the start. This leads to a very large memory footprint at the start because usually, all data present is used to construct an observation tree (a PTA, prefix tree acceptor). Solutions for this problem could be sampling the initial data, or learning the automata using a streaming approach (Balle et al., 2012, 2014; Baumgartner and Verwer, 2022, 2023; Schmidt and Kramer, 2014).

In this paper we present the algorithm DAALder that is a combination of active and passive learning. Instead of loading all the available data in memory, the data is saved on disk in a database. Databases allow for efficient querying of the data and frequently is the original data source. Especially when learning from software logs, the logging information is frequently stored in a large distributed database, for example, the very popular Splunk database[1]. We develop a type of active learning algorithm that, instead of asking membership and equivalence queries, asks database queries to iteratively construct a state machine.

A key benefit of our approach is that membership queries in traditional active learning ask for a very specific set of traces. It is likely that several of the asked traces are not present in the collected data. In this case, it is unclear how to proceed. There is research on active learning algorithms in a setting where not all queries can be answered. In these settings, the teacher, the component that answers the query, is referred to as an inexperienced teacher (Leucker and Neider, 2012) or incomplete teacher (Moeller et al., 2023). These approaches rely on SAT solving (Grinchtein and Leucker, 2006; Moeller et al., 2023), use containment queries (Chen et al., 2009), or loosen L* with weakened concepts (Grinchtein et al., 2006). All these approaches have the CSP (constraint-satisfaction problem) (Biermann and Feldman, 1972) as a basis for their own approach (Leucker and Neider, 2012). Relying on SAT solvers, however, can take a lot of time (Moeller et al., 2023).

Intuitively, our approach avoids asking for this specific set of traces, which eventually forms a characteristic sample (de la Higuera, 2010). For passive automaton learning algorithms, the presence of such a sample in the data implies that greedy algorithms such as RPNI (Oncina and Garcia, 1992) are guaranteed to find a model for (converges to) the target language (that is assumed to have generated the data). There exist many possible characteristic samples, but traditional active learning only constructs a specific one. Our main goal is to use a combination of active and passive learning such that the algorithm converges to the target when any characteristic sample is present in the database, while observing only a fraction of the data. Our contributions are:

- A new algorithm, named DAALder, to learn a state machine from a database of traces.
- An implementation[2] of this algorithm in the state merging software package FlexFringe (Verwer and Hammerschmidt, 2017, 2022).
- Results on the performance of DAALder compared with a conventional EDSM (Evidence-Driven State Merging) algorithm on datasets of increasing size.

2. Algorithm Description

2.1. Queries

For conventional active learning, the membership query and the equivalence query exist. In practice, it is not feasible to perform an equivalence query exactly. Often, a randomized search is employed, which is also done for this implementation. (See Appendix A.1.) Membership queries are used to grow the model. A membership query returns given an input sequence (a trace) its output symbol. However, using a database enables us the freedom to design queries that are more powerful than membership queries, yet easy to compute.

The novel query used in this algorithm is the prefix query. This query is noted as PrefixQuery(t, n, k) with t being the prefix trace, n being the max size of the returning traces, and k the amount of traces to return. This query returns up to a number of k traces

1. `https://www.splunk.com/`
2. Available after acceptance of the paper in `https://github.com/tudelft-cda-lab/FlexFringe`

alongside their output symbol that has t as their prefix. Parameter n can limit the result by their length to explore only a specific depth. Implementation details in Appendix A.2.

2.2. Algorithm: DAALder

DAALder, the algorithm presented here, is a form of a state merging algorithm in the red-blue-framework (Lang et al., 1998). However, instead of constructing a PTA directly from all the data available, it is constructed iteratively using the at that moment available traces from the complete data that the algorithm deemed informative.

$L^{\#}$ (Vaandrager et al., 2022a,b) was a large inspiration for DAALder. Just like the $L^{\#}$ algorithm, DAALder works directly on a PTA data structure (or observation tree). This gives rise to an efficient way to search for new states and generate hypotheses.

2.2.1. Outline

Pseudocode and an outline of DAALder are found in algorithm 1 and in figure 1, respectively.

DAALder uses a state merging routine similar to $L^{\#}$. It performs multiple rounds of state merging, each time with more data. Occasionally, it will present a hypothesis to the oracle in the hopes of providing the correct hypothesis.

From EDSM, we know that information on the traces that pass through nodes can be used to determine whether two nodes are equivalent and, thus, can be merged. Given some scoring heuristics, merges can be compared, and when a merge has the largest score, the merge is likely merging equivalent states (Lang et al., 1998).

DAALder uses these exact scoring heuristics to, next to selecting merges, also guide what queries it should ask. If during merging, for the same blue node, merges with a similar score are encountered, those merges are put on hold. Instead of selecting one of these uncertain merges, it asks for more data for the nodes involved in them and thus providing the inconclusive nodes with more information. (in DAALderProcessUnidentified)

DAALder will then ask prefix queries for the nodes involved in these uncertain merges. Traces returned by these queries will contain more information for the scoring heuristics when calculating scores again for those nodes. Traces gained by this are added to the observation tree whenever a new merging routine is started. This new data can then be used to be more certain about merges to perform in the new round that previously were doubtful.

After merging, DAALder can either present its hypothesis to the equivalence oracle or reset the tree and start a new round of merging. If there had been a lot of new information, it is wise just to reset the tree, because a lot of new information might imply the current hypothesis is wrong and the oracle takes much time. If presented to the oracle, the oracle can answer ”yes” or present a counterexample to be processed, and the tree is also reset.

In the end, when it finds a sufficiently correct state machine, it has gathered a subset of traces that contain enough information to construct a correct hypothesis. This is a so-called characteristic set of traces (de la Higuera, 2010), or tell-tale set (Angluin, 1980).

DAALder is thus a combination of active and passive learning. It performs a search strategy to find new information based on the state-of-the-art active learning approach from $L^{\#}$ and the highly flexible state-of-the-art implementation of passive learning state merging algorithms from FlexFringe. This also means that all variations and hyperparameters on state merging available in FlexFringe are also directly available to DAALder.

Algorithm 1: DAALder algorithm

```
Data: A set of traces with an output label
Result: A hypothesis $\mathcal{H}$ consistent with the provided traces
Initialize: $\mathcal{T}$ a PTA (observation tree) with red core $S$ and blue fringe $F$
Initialize: $S$ red core nodes, initially with a root node
Initialize: $F$ blue fringe nodes, initialize using some random traces
Loop
    state_isolated ← $false$
    unidentified ← $\varnothing$ (The blue states without merge candidate)
    all_merges_todo ← $\varnothing$ (Merges to do, to make the hypothesis state machine)
    foreach $p \in F$ do
        possible_merges ← 0
        merge_candidate ← ?
        foreach $q \in S$ do
            if CheckConsistency($p$, $q$) then
                possible_merges ← possible_merges + 1
                merge_candidate ← $q$
            end
        end
        if possible_merges = 1 then
            all_merges_todo add($p$, merge_candidate) (identified node)
        end
        if possible_merges = 0 then
            DAALPromote($p$)
            state_isolated ← $true$
        end
        if possible_merges > 1 then
            unidentified add $p$ (Investigate these states more)
        end
    end
    (explore_more, merges_todo) ← DAALProcessUnidentified(unidentified)
    all_merges_todo ← all_merges_todo ∪ merges_todo
    if explore_more or state_isolated then
        continue (Investigate the added traces and/or a new fringe/frontier)
    end
    $\mathcal{H}$ ← BuildHypothesis($\mathcal{T}$, all_merges_todo)
    $\sigma$ ← EquivalenceOracle($\mathcal{H}$)
    if $\sigma$ then
        ProcessCounterExample($\mathcal{T}$, $\sigma$)
        continue (Did not found the solution yet)
    end
    return $\mathcal{H}$ (Solution found!)
end
```

DAALProcessUnidentified and ProcessCounterExample add new traces to the observation tree. In our implementation DAALProcessUnidentified asks for more information on nodes when the highest merge score is less than twice the second-largest merge score.

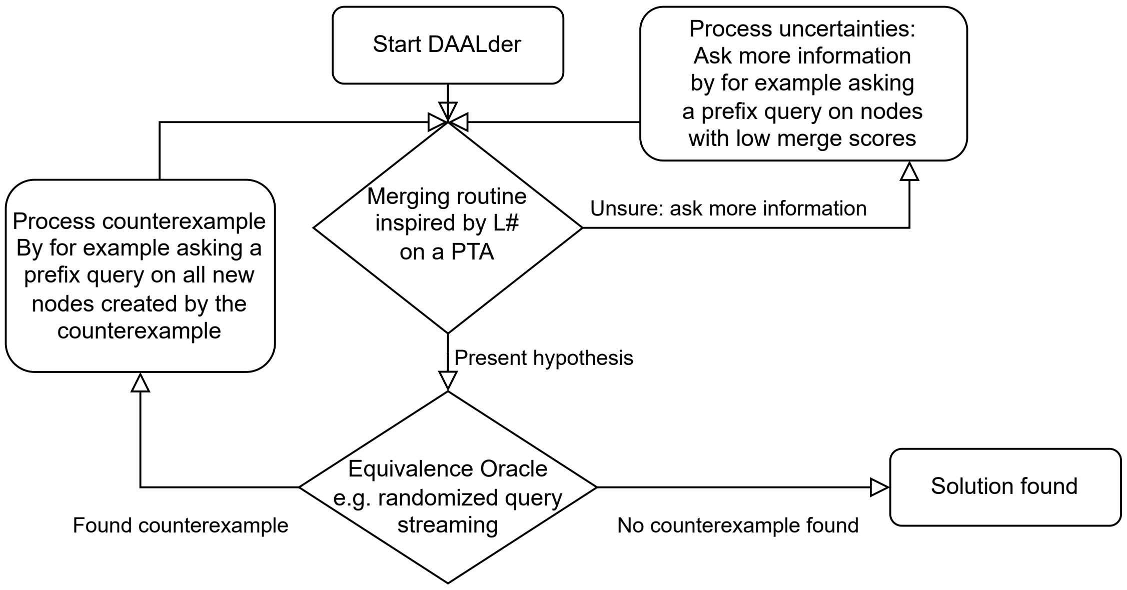

Start DAALder
Process uncertainties: Ask more information by for example asking a prefix query on nodes with low merge scores
Process counterexample By for example asking a prefix query on all new nodes created by the counterexample
Merging routine inspired by L# on a PTA
Unsure: ask more information
Present hypothesis
Equivalence Oracle e.g. randomized query streaming
Found counterexample
No counterexample found
Solution found

Figure 1: Graphical representation of DAALder

2.2.2. Explore vs exploit

Besides the choice of state merging heuristics, DAALder provides another way to tweak its inner workings. This is the DAALProcessUnidentified subroutine. Here, a choice must be made between adding more data to combat uncertainty or being more daring and asking the equivalence oracle earlier. This is a choice between exploration (by asking for more data) and exploitation of the current data (by asking the equivalence oracle). Asking the equivalence oracle could be a time-consuming process, but so could be asking for more data, in addition to the memory required to add the data to the observation tree.

3. Algorithm Evaluation

3.1. Data generation

Artificial trace data has been generated by a modified version of `FSM-learning` from Giantamidis et al. (2021) available at https://github.com/hwalinga/FSM-learning. The program is modified to use JSON serializing instead, as its binary format did not work on Linux. It is also modified to use a streaming approach to output data instead of writing it at the end of the execution of the program to reduce memory requirements.

Artificial state machines were generated with `gen_moore.py`. The input and output alphabet were both set to a size of two. The number of states was set to two hundred. Giantamidis et al. (2021) took inspiration from (Lang et al., 1998) for this program.

The `generate` program was used to generate traces. The program generates the traces using a random walk and maintains frequencies of states visited before which it uses to calculate probabilities of states to visit next during its random walk.

Every single dataset had its own randomized state machine. The sizes for the datasets varied from 625 unique traces to 40.960.000 unique traces for the train data. The test set had a fixed amount of traces of 1.000.000.

3.2. Experiments

The DAALder algorithm is compared with the FlexFringe implementation of EDSM (Evidence Driven State Merging). For configuration see Appendix B. EDSM is run with low memory (capped at 16GB with cgroups) and high memory (70GB). These tests show EDSM's response to using swap memory. To test the exploration-vs-exploitation balance for DAALder, its hyperparameter k (see section 2.1) was set at 10, 100, and 1000.

Memory usage in the experiments was measured as virtual memory. This includes the swap memory. The memory was measured using `ps -U postgres -o vsz` while running the algorithms under the *postgres* user as well. This way, we also include any memory allocated by the database engine itself.

4. Results

Figure 2 and 3 show the results of the experiments. Figure 2 depicts the relationships between the amount of traces versus the time the algorithm took to finish and the memory used by the algorithm, Figure 3 depicts the traces eventually included in the model (for DAALder) and the percentage that this is of all the traces in the dataset.

Both EDSM and DAALder with the different parameters scored equally well on accuracy. The algorithms scored perfectly for 80k traces or higher, and for the datasets with 40k and lower, they scored just as good as guessing (50%).

Some data is missing because for EDSM the biggest datasets ran out of memory (even using swap), trying to allocate more than 100 GB. For DAALder data is missing around the 40k data points mark because running it took over 10+ hours and we choose to kill the program. There are also no data points for memory reading for EDSM for the smallest datasets, as the program was too quickly done for a memory measurement to finish.

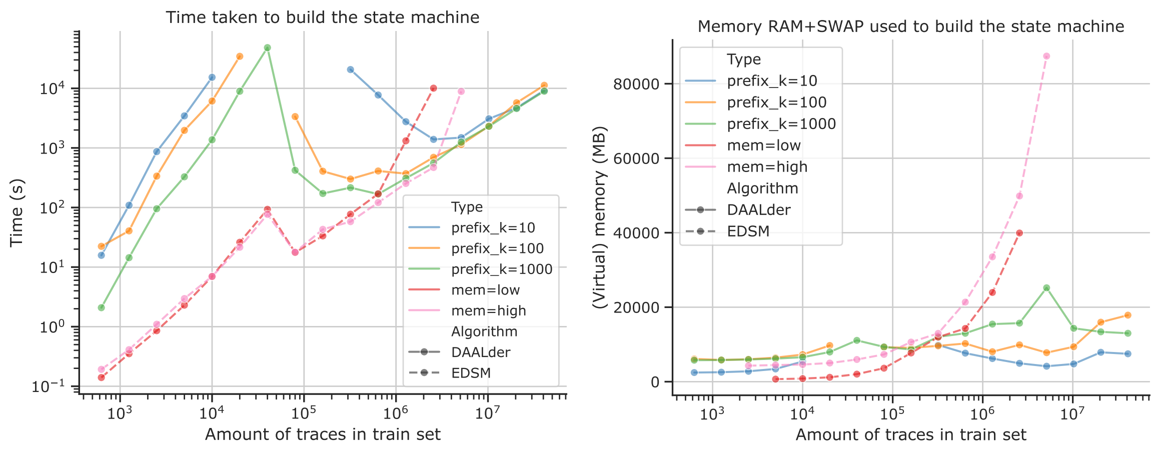


Figure 2: Time and memory usage for EDSM vs DAALder

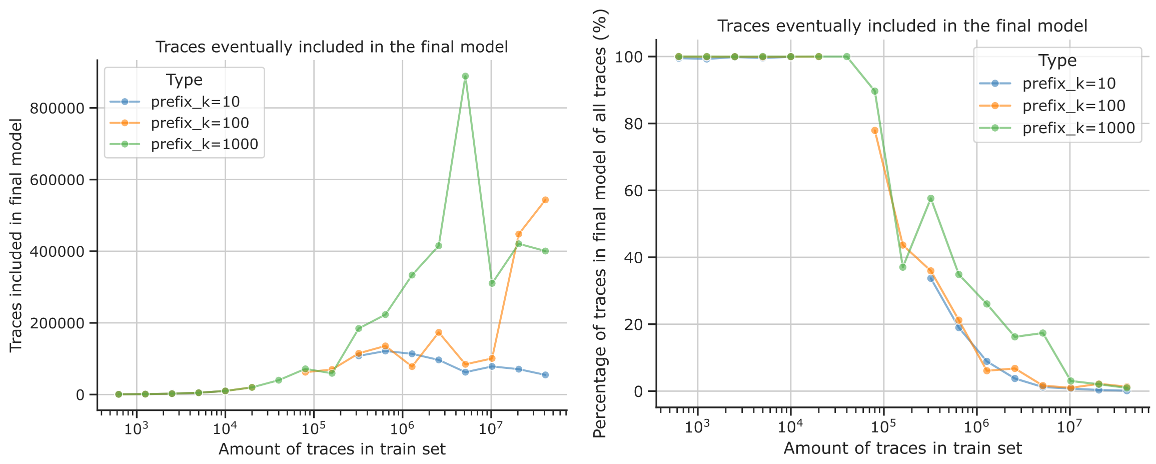


Figure 3: Amount of traces included in the final model for DAALder

5. Discussion

5.1. Solvability point

As remarked earlier, there was a clear tipping point in the experiments based on the accuracy scores. After the 40k data mark, all algorithms scored 100% on accuracy; before that, they were just as good as just guessing the output class. This means that the sampled data is after this point now big enough to contain a characteristic set (de la Higuera, 2010). This amount of data seems to be the solvability tipping point for the algorithms on these kinds of datasets.

After the solvability point, the EDSM algorithm's runtime drops briefly. It can now make the correct automaton, whereas before, it was heavily overfitted on the sparse data it had to work with, reporting a hypothesis with many more states than necessary. It can also be seen in the log files that there is no ambiguity in merges to be made anymore. That is, after the solvability point, there is always only one possible merge.

Around the solvability tipping point, DAALder performs the worst in terms of runtime. It took so long to finish that it had to be shut down before finishing. As there is not enough data yet to arrive at a solution, DAALder iteratively keeps including more data until (almost) all data is included. This takes a lot of time. This can also be seen in figure 2, where there is a clear peak around this point at the 40.000 traces mark.

5.2. Memory usage

When EDSM runs out of RAM memory, it resorts to swap. This behavior is very visible for the large datasets between the low and the high memory setups. When EDSM had to resort to swapping, its runtime increased more steeply. The slope for this increasing runtime was also significantly more than for DAALder for the largest datasets. However, it seems that EDSM would perform on par with DAALder if given enough RAM.

The memory usage of EDSM increases exponentially. So much so that not all experiments could be completed for EDSM. For DAALder, the memory usage remained equal at just a few GBs. This can be explained by the fact that DAALder ensures that just enough traces are included to create a correct model. In fact, when k was set to 10, the by DAALder selected set of traces was around the same size as the sample size around the solvability tipping point. Thus, if explore settings are not too aggressive, DAALder is indeed able to select a set just large enough of the most informative traces to represent the state machine it tries to learn.

5.3. Finding a characteristic set

It is important to note that these tests were run on data from a statistical uniform sampler. The effect of this is that for a random sample, the minimum size that contains a characteristic set is much smaller, than if the data was generated with a non-uniform sampler. If some transitions of the automaton are only described by a set of rare traces, these are less likely to be found in random samples of this data. For those datasets, sampling data would not be a viable strategy.

For our uniform sampled data, DAALder used a set of traces that is somewhat equal in size to the sample size at a solvability tipping point. However, for a different dataset with traces that traverse its state machine non-uniformly, we expect that DAALder is able to find a much smaller characteristic dataset, than the sample size that would be required if it was randomly sampled.

5.4. Effect of k

Increasing k (including more traces for each prefix query) for DAALder had the effect of also ending up with more traces in the end (and thus also more RAM). For the sparse datasets, the runtime decreased with increasing k. However, when the datasets grow larger, and sparsity decreases, the runtimes for different values of k are very close to each other. It seems that if sparsity decreases, a smaller set of traces from a prefix query is just as well able to distinguish a state as a larger set of traces from a prefix query.

6. Conclusions and Future Work

This paper presents DAALder, a new algorithm combining previous work on active learning and passive learning. It shows that this algorithm is faster than conventional state merging algorithms, when these algorithms do not have enough RAM memory for very large datasets.

Future work requires checking how well DAALder performs on different kinds of datasets and state machines. It will focus on researching different approaches to the explore-exploit balance, and finding what works best on these different datasets, and eventually learning from real-life datasets.

What is also of interest, is if we can increase our toolbox of queries. Designing different kinds of queries that are optimized by different database technologies, opens up more possible state machine learning algorithms that operate on trace databases.

One of these is to perform an equivalence query directly on the database. Although regular expressions can be executed on databases, these implementations do not scale. Furthermore, a consistency test can also directly be evaluated on the database for two states by comparing the prefix queries of the two access traces with each other.

Appendix A. Implementation details

A.1. Randomized query

Instead of checking a random primary key one by one, this can be done much more efficiently by using a pseudo-random streaming approach using libpqxx based on the PostGreSQL COPY protocol. (https://web.archive.org/web/20231212092242/https://pqxx.org/development/libpqxx/#2023-01-12-at-last-faster-than-c)

A.2. Prefix query

The prefix query is implemented as follows:

```sql
SELECT * FROM table WHERE trace LIKE :'t' || '%' AND LENGTH(trace) < :n LIMIT :k;
```

Snippet 1: Select k traces that start with t and are shorter than n.

The prefix query can be sped up using the SP-GiST indexing, which is a similar data structure to a PTA, but optimized for disk usage (Eltabakh et al., 2006).

Appendix B. EDSM parameters

EDSM is run with parameters defined in `ini/edsm.ini`. This specifies that there are no sink nodes and there is no lower bound. In addition, only the blue node that currently has the most amount of traces following it, is considered for a merge, or a promotion to a red node if no merges are available to it.

Appendix C. Setup

The tests are run on a Dell PowerEdge R710 with an Intel Xeon CPU E5620, Seagate ST3146356SS, and Hynix HMT151R7BFR4C-H9 memory sticks. Memory speed was measured with `sysbench` as well as the hard disk read speed with mode `rndrd`. The CPU speed was measured under `stress` from `/proc/cpuinfo`. The performance is a memory speed of 4100 MiB/s, an HDD random read speed of 2.7 MiB/s, and a CPU speed of 2527 MHz. The computer ran with Ubuntu 22 and postgresql 14.11, libpqxx 7.9.0, and clang 16.0.6.

References

Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *ACM SIGPLAN Notices*, 37(1):4–16, January 2002. ISSN 0362-1340. doi: 10.1145/565816.503275. URL https://dl.acm.org/doi/10.1145/565816.503275.

Dana Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45(2):117–135, May 1980. ISSN 0019-9958. doi: 10.1016/S0019-9958(80)90285-5. URL https://www.sciencedirect.com/science/article/pii/S0019995880902855.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987. ISSN 08905401. doi: 10.1016/0890-5401(87)90052-6. URL https://linkinghub.elsevier.com/retrieve/pii/0890540187900526.

Joao Antunes, Nuno Neves, and Paulo Verissimo. Reverse Engineering of Protocols from Network Traces. In *2011 18th Working Conference on Reverse Engineering*, pages 169–178, Limerick, Ireland, October 2011. IEEE. ISBN 978-1-4577-1948-6. doi: 10.1109/WCRE.2011.28. URL http://ieeexplore.ieee.org/document/6079839/.

Borja Balle, Jorge Castro, and Ricard Gavaldà. Bootstrapping and Learning PDFA in Data Streams. In *Proceedings of the Eleventh International Conference on Grammatical Inference*, pages 34–48. PMLR, August 2012. URL https://proceedings.mlr.press/v21/balle12a.html.

Borja Balle, Jorge Castro, and Ricard Gavaldà. Adaptively learning probabilistic deterministic automata from data streams. *Machine Learning*, 96(1):99–127, July 2014. ISSN 1573-0565. doi: 10.1007/s10994-013-5408-x. URL https://doi.org/10.1007/s10994-013-5408-x.

Robert Baumgartner and Sicco Verwer. Learning state machines via efficient hashing of future traces, July 2022. URL http://arxiv.org/abs/2207.01516.

Robert Baumgartner and Sicco Verwer. Learning state machines from data streams: A generic strategy and an improved heuristic. In *Proceedings of 16th Edition of the International Conference on Grammatical Inference*, pages 117–141. PMLR, July 2023. URL https://proceedings.mlr.press/v217/baumgartner23a.html.

Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 141–150, New York, NY, USA, August 2009. Association for Computing Machinery. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595719. URL https://dl.acm.org/doi/10.1145/1595696.1595719.

A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Transactions on Computers*, C-21(6):592–597, June 1972. ISSN 1557-9956. doi: 10.1109/TC.1972.5009015. URL https://ieeexplore.ieee.org/abstract/document/5009015.

Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning Minimal Separating DFA's for Compositional Verification. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00767-5 978-3-642-00768-2. doi: 10.1007/978-3-642-00768-2_3. URL http://link.springer.com/10.1007/978-3-642-00768-2_3.

Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. {MACE}: {Model-inference-Assisted} Concolic Exploration for Protocol and Vulnerability Discovery. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011. URL https://www.usenix.org/conference/usenix-security-11/mace-model-inference-assisted-concolic-exploration-protocol-and.

Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol Specification Extraction. In *2009 30th IEEE Symposium on Security and Privacy*, pages 110–125, May 2009. doi: 10.1109/SP.2009.14. URL https://ieeexplore.ieee.org/abstract/document/5207640.

Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998. ISSN 1049-331X. doi: 10.1145/287000.287001. URL https://dl.acm.org/doi/10.1145/287000.287001.

Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic Protocol Reverse Engineering. *USENIX Security Symposium*, pages 1–14, 2007.

C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. CUP, 2010.

M.Y. Eltabakh, R. Eltarras, and W.G. Aref. Space-Partitioning Trees in PostgreSQL: Realization and Performance. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 100–100, April 2006. doi: 10.1109/ICDE.2006.146.

Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017, pages 142–151, New York, NY, USA, July 2017. Association for Computing Machinery. ISBN 978-1-4503-5077-8. doi: 10.1145/3092282.3092289. URL https://dl.acm.org/doi/10.1145/3092282.3092289.

Georgios Giantamidis, Stavros Tripakis, and Stylianos Basagiannis. Learning Moore machines from input–output traces. *International Journal on Software Tools for Technology Transfer*, 23(1):1–29, February 2021. ISSN 1433-2787. doi: 10.1007/s10009-019-00544-0. URL https://doi.org/10.1007/s10009-019-00544-0.

Olga Grinchtein and Martin Leucker. Learning Finite-State Machines from Inexperienced Teachers. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, *Grammatical Inference: Algorithms and Applications*, Lecture Notes in Computer Science, pages 344–345, Berlin, Heidelberg, 2006. Springer. ISBN 978-3-540-45265-2. doi: 10.1007/11872436_30.

Olga Grinchtein, Martin Leucker, and Nir Piterman. Inferring Network Invariants Automatically. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Ulrich Furbach, and Natarajan Shankar, editors, *Automated Reasoning*, volume 4130, pages 483–497. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-37187-8 978-3-540-37188-5. doi: 10.1007/11814771_40. URL http://link.springer.com/10.1007/11814771_40.

Andreas Hagerer, Tiziana Margaria, Oliver Niese, Bernhard Steffen, Georg Brune, and Hans-Dieter Ide. Efficient Regression Testing of CTI-Systems. *Annual review of communication*, 55:1033–1040, 2001.

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 Vasant Honavar and Giora Slutzki, editors, *Grammatical Inference*, Lecture Notes in Computer Science, pages 1–12, Berlin, Heidelberg, 1998. Springer. ISBN 978-3-540-68707-8. doi: 10.1007/BFb0054059.

Martin Leucker and Daniel Neider. Learning Minimal Deterministic Automata from Inexperienced Teachers. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, Lecture Notes in Computer Science, pages 524–538, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-34026-0. doi: 10.1007/978-3-642-34026-0_39.

Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva. Automata Learning with an Incomplete Teacher. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.21. URL https://drops.dagstuhl.de/opus/volltexte/2023/18214.

Jos'e Oncina and Pedro Garcia. Identifying regular languages in polynomial time. *Advances in Structural and Syntactic Pattern Recognition*, pages 99–108, 1992. URL https://www.worldscientific.com/doi/abs/10.1142/9789812797919_0007.

Jana Schmidt and Stefan Kramer. Online Induction of Probabilistic Real-Time Automata. *Journal of Computer Science and Technology*, 29(3):345–360, May 2014. ISSN 1860-4749. doi: 10.1007/s11390-014-1435-8. URL https://doi.org/10.1007/s11390-014-1435-8.

Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 223–243, Cham, 2022a. Springer International Publishing. ISBN 978-3-030-99524-9. doi: 10.1007/978-3-030-99524-9_12.

Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness, January 2022b. URL http://arxiv.org/abs/2107.05419.

Sicco Verwer and Christian Hammerschmidt. FlexFringe: Modeling Software Behavior by Learning Probabilistic Automata, 2022. URL http://arxiv.org/abs/2203.16331.

Sicco Verwer and Christian A. Hammerschmidt. Flexfringe: A Passive Automaton Learning Package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642, September 2017. doi: 10.1109/ICSME.2017.58. URL https://ieeexplore.ieee.org/abstract/document/8094471.

CHAPTER 2

Background

2.1 State machines

State machines are also known as finite state machines (FSM) or finite state automata (FSA). These are formally defined as a set representing the alphabet, a set of states, a function that shows an output for all states, and a function that shows the transitions for input. Variations exist for how the output is constructed with the state machine. For a finite state machine, the only two outputs are accepting and rejecting outputs based on what state the state machine is currently in. For a Moore machine, the output of the current state can be anything. In comparison, for a Mealy machine, the output is not only dependent on the current state but also on the previous transition [15].

The finite state machine can be mathematically represented as the tuple $(Q, I, \delta, \lambda, O)$ with:

1. Q is the set of states, including the initial state $q_0 \in Q$.
2. I is the alphabet of the input sequence. Where I^* is used to indicate the set of all finite strings that can be constructed from this alphabet.
3. δ is the transition function that maps the state transitions given the input: $\delta : Q \times I \to Q$.
4. λ is the output function that maps each state to an output: $\lambda : Q \to O$.
5. O is the set of all possible outputs of each state.

Note that for a simple FSM, the output is just accepting or rejecting thus, O contains only 2 items, as opposed to the Moore machine, where O can be of arbitrary size. For a Mealy machine, the transition function and output function are combined, as the output is also dependent on the last input, thus $\langle \delta, \lambda \rangle : Q \times I \to O \times Q$.

State machines can be divided into two types, the deterministic one and the non-deterministic one, abbreviated with DFA (deterministic finite-state machine) and NFA (non-deterministic finite-state

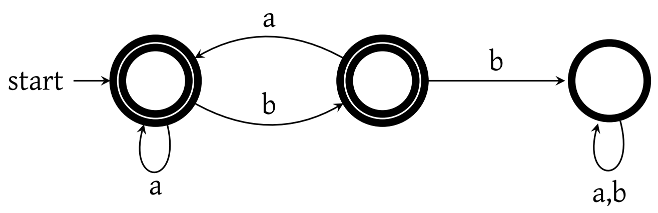


Figure 2.1: An example of a state machine that rejects strings with two consecutive b's. The double border indicates an accepting state, and the order state is rejecting. Only when two consecutive symbols are both b, a rejecting state is reached, and you remain there.

machine). For a state machine to be identified as a deterministic one, all of its transitions must be uniquely defined by the source state and the inputs that follow after arriving at the source state. An NFA can be transformed into a DFA, but that would result in exponentially more states. For the purpose of this thesis, only deterministic machines are considered.

Augmenting weights to the transitions of a state machine gives rise to a probabilistic deterministic finite-state automaton (PDFA). The weights denote the probability of a transition and can be used, for example, to identify rare transitions within a system. The output of a PDFA is no longer a set but a function that can assign probability to a set of transitions instead of either rejecting or accepting it. This allows us to determine if a system is experiencing a very rare set of transitions but not an impossible one.

2.2 Inference

A pivotal concept in these learning algorithms is the concept of Myhill–Nerode congruence. The Myhill-Nerode congruence signifies that if two starting strings in a regular language have no extensions that can distinguish the two, then these two strings must be *equivalent* in the context of the language. This means that they end in the same state in the corresponding state machine. We can also state that all future responses will be the same for two "states" that are Myhill-Nerode equivalent (or simply equivalent). We also say that these strings are then the *access* traces for these states. The Myhill-Nerode theorem further states that if all the strings can be divided into a finite set of equivalence classes, the language for those strings is regular. If all the equivalence classes are then found, all the states for the corresponding state machine are also found, establishing a bijection between regular languages and their corresponding minimal state machines [16]. During inference of the state machines, in essence these equivalence states are the ones that are searched for.

2.2.1 Active learning

MAT

The most prevalent framework for active learning defines the SUL as the minimally adequate teacher (MAT). This MAT can only answer the result of a trace from a system (a membership query) or if the current hypothesis correctly represents the system (an equivalence query). With these different kinds of queries alone, the model can be learned in a polynomial amount of queries [17]. However, the equivalence query is not trivial to implement and, in practice, will often take a lot of time to answer.

L^* algorithm

The first developed active learning algorithm is the L^* algorithm. It iteratively builds and extends an observation table, which can be used to construct a state machine.

The observation table keeps a set of strings as rows and columns. When building the automaton, the rows represent the access traces of the states (**STA**), and the columns represent the experiment set (**EXP**). The latter are extensions to the access traces, and they are used to find equivalent strings.

When the content of the table is filled, you can define two different sets of rows of the **STA**. The content of the table is the result of the query combined from the access trace and the extension sequence (**OT** : **STA** $\times$ **EXP**). The access traces can then be divided into the RED and BLUE states, where the RED states all have different results for the extensions and thus are the different states in the resulting state machine. The BLUE states are states that have a state in the set of RED states that have the same outcome for the experiments. When the results in the row for two access traces are the same, these are called *equivalent*.

Here follows a brief overview of the L^* algorithm highlighting some differences from the original as presented by Angluin [17]. L^* originally was designed just for learning FSM, but with some adjustments, it can also learn Mealy machines [19].

First, the observation table must be closed. That means that for every BLUE state, there must be an equivalent RED state. If this is not the case, that BLUE state must be promoted to be a RED state. Then, add all the single-letter extensions of the promoted access trace to the set of BLUE states.

	λ
λ	1
a	1
ab	
abb	0
b	1
aa	
aba	
abba	
abbb	

(a) Table after equivalence query returned abb (as not in L).

	λ
λ	1
a	1
ab	1
abb	0
b	1
aa	1
aba	1
abba	0
abbb	0

(b) Membership queries are made: Table is not closed.

	λ	b
λ	1	1
a	1	1
ab	1	0
abb	0	0
b	1	
aa	1	
aba	1	
abba	0	
abbb	0	

(c) Adding a column to make the table closed.

	λ	b
λ	1	1
a	1	1
ab	1	0
abb	0	0
b	1	0
aa	1	1
aba	1	1
abba	0	0
abbb	0	0

(d) The table after filling the holes is closed and consistent.

Figure 2.2: An iteration on an observation table and the resulting state machine. It learns the state machine from figure 2.1. The λ indicates the empty string. Image courtesy to de la Higuera [18]

Next, the observation table must be consistent. That means that for every pair of Red states, the possible extensions must also be equivalent. Thus if $OT[s_1] = OT[s_2]$ but $OT[s_1 a][e] \neq OT[s_2 a][e]$, then we know that ae must be added to the experiment set.

Finally, when the equivalence query presents a counterexample, this must be processed. In the original paper presenting L^*, all the prefixes of the counterexample were added to the Red set. However, by utilizing some clever counterexample processing introduced in later adaptations of the L^* algorithm, only the exact data that makes the hypothesis wrong can be added to the observation table, which can also take into account the consistency check directly. In the classical L^* learning, prefixes of the counterexample were also added to the Red states, but by keeping only equivalent classes in the Red states, any inconsistencies are not introduced. For example in Shahbaz and Groz [19] a method is proposed that splits the counterexample by removing the longest known prefix that can be found in the current access traces and adding all suffixes of the remaining trimmed suffix to the experiment set. Another example in Rivest and Schapire [20] uses binary search to split the counterexample iteratively until the specific sequence is found that makes the hypothesis deviate from the system behavior. It then again adds all the suffixes of the remaining sequence to the experiment set.

The L^* algorithm is outlined in algorithm 1.

Algorithm 1: L^* algorithm

```
1  Loop
2      Loop
3          if ObservationTable not closed then
4              LStarClose(ObservationTable)
5          else if ObservationTable not consistent then
6              LStarMakeConsistent(ObservationTable)
7          else
8              break (to make hypothesis)
9          end
10     end
11     Hypothesis ← LStarMakeHypothesis(ObservationTable)
12     CounterExample ← EquivalenceOracle(Hypothesis)
13     if CounterExample then
14         LStarProcessCounterExample(CounterExample)
15     else
16         return (found solution)
17     end
18 end
```

2.2.2 Passive learning

State merging

The prevalent method in passive learning is state merging. In a state merging algorithm first, a(n) (augmented) prefix tree acceptor ((A)PTA) with all the positive traces is constructed, where each prefix ends in a state [21]. The APTA is a tree that describes the input traces without any loops or re-convergent paths. The subsequent step involves merging nodes of the tree that are most likely to be from two equivalent states given an evidence-driven heuristic. During merging, the tree will evolve into a smaller and smaller graph. The nodes in this graph represent the equivalence states of the state machine. This merging continues until all the states of the state machine are identified. This is often referred to as EDSM (evidence-driven state merging). This method traces its roots back to the RPNI algorithm [22].

Blue-fringe algorithm

The blue-fringe algorithm, also known as the red-blue framework, is a state merging strategy that emerged during a state machine learning competition [23]. This strategy is an improvement on the idea of EDSM.

The outline of this algorithm is presented in algorithm 2.

The difference between the blue-fringe strategy and previous EDSM approaches is that not all possible merges are considered. Instead, a node coloring strategy is employed to limit the set of merges

Algorithm 2: Blue-fringe algorithm

```
1  Create PTA from traces
2  Color root of PTA RED
3  Color children of root BLUE
4  repeat
5      scores ← $\varnothing$
6      foreach $p \in$ BLUE do
7          foreach $q \in$ RED do
8              if CONSISTENTCHECK(PTA, $p$, $q$) then
9                  scores[$p$, $q$] ← CALCULATESCORE(PTA, $p$, $q$)
10             end
11         end
12     end
13     if scores = $\varnothing$ (no consistent merges) then
14         Color a BLUE node RED. Color its children BLUE
15     else
16         Perform the highest scoring MERGE between $p$ and $q$ in scores.
17     end
18 until No BLUE nodes left;
```

considered. Here, the root of the tree is first colored red, and all the children of a red node are colored blue. All other nodes remain uncolored or simply white.

Now, iteratively, a node from the core red set and one from the blue fringe set that are suspected to be equivalent are merged. Which nodes are merged are again determined by a certain heuristic, providing scores for the most likely equivalent nodes. A consistency check ensures that only merges that do not alter the outcome of the state machine are taken into account. If a blue node cannot be merged with any red node, the blue node is promoted to be a red node as well, coloring all its children blue again.

Whenever a node is merged, it is possible that subsequent merges must be executed to make the state machine deterministic again. This is called folding [18] or determinization [24, 25]. This process is where a scoring heuristic can be defined. This can, for example, be all the labels that undergo folding and end up being correct as defined in the original Abbadingo algorithm [23], or all the nodes that undergo the determinization and end up being correct as in FlexFringe [24, 25].

Notably, this framework can be extended to learn probabilistic state machines, and many different merging heuristics can be implemented in this framework. These will be further expanded in the state-of-the-art section (see section 2.4.1). Note that nowadays the node coloring heuristic for this algorithm as presented in Lang, Pearlmutter, and Price [23] is sometimes just referred to as EDSM [26].

Example Let's learn the state machine from figure 2.1 with the data:
$< 1, a >, < 1, aa >, < 1, ba >, < 0, bb >, < 0, abb >, < 1, bab >$.
First, construct the APTA as seen in figure 2.3. (Dashed border indicates an unknown label.)

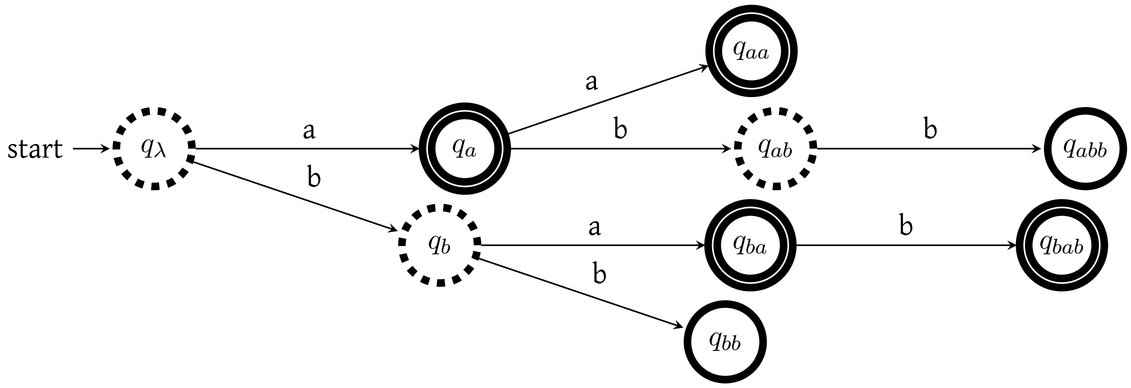


Figure 2.3: The APTA for the data: $< 1, a >, < 1, aa >, < 1, ba >, < 0, bb >, < 0, abb >, < 1, bab >$. The root with the adjacent nodes has no label associated with them.

Now a merging is possible between $< q_\lambda, q_a >$ and $< q_\lambda, q_b >$. Both the merges yield a consistent (intermediate) automaton. However, when counting the number of subsequent merges during the determinization process, $< q_\lambda, q_a >$ make more subsequent merges. This is the merge between q_{abb} and q_{bb} and the merge between q_a and q_{aa}. For the merge $< q_\lambda, q_b >$, the only subsequent merge that is counted is between q_a and q_{ba}. (Merges with nodes that do not have a label do not count.) This gives the merge $< q_\lambda, q_a >$ a higher score in an EDSM algorithm and is thus the preferred merge. Let's continue with both merges to see the end result. These are worked out on the next two pages.

Both continuations yield two different automata. They both describe the data correctly but have a different structure. The result of $< q_\lambda, q_b >$ is, however, a bit more complex. It has more edges and also does not have the sink state q_{bb} (a state with no outgoing edges). This is eventually the goal of EDSM. Creating an automaton that is relatively simple. It does not promise the simplest or smallest automaton. It is just a heuristic to get reasonable results.

The intuition behind this heuristic is that if you are able to merge relative complex structures with each other (thus gain a large merge score), this merge is the best one to take to result in a reasonable simple automaton.

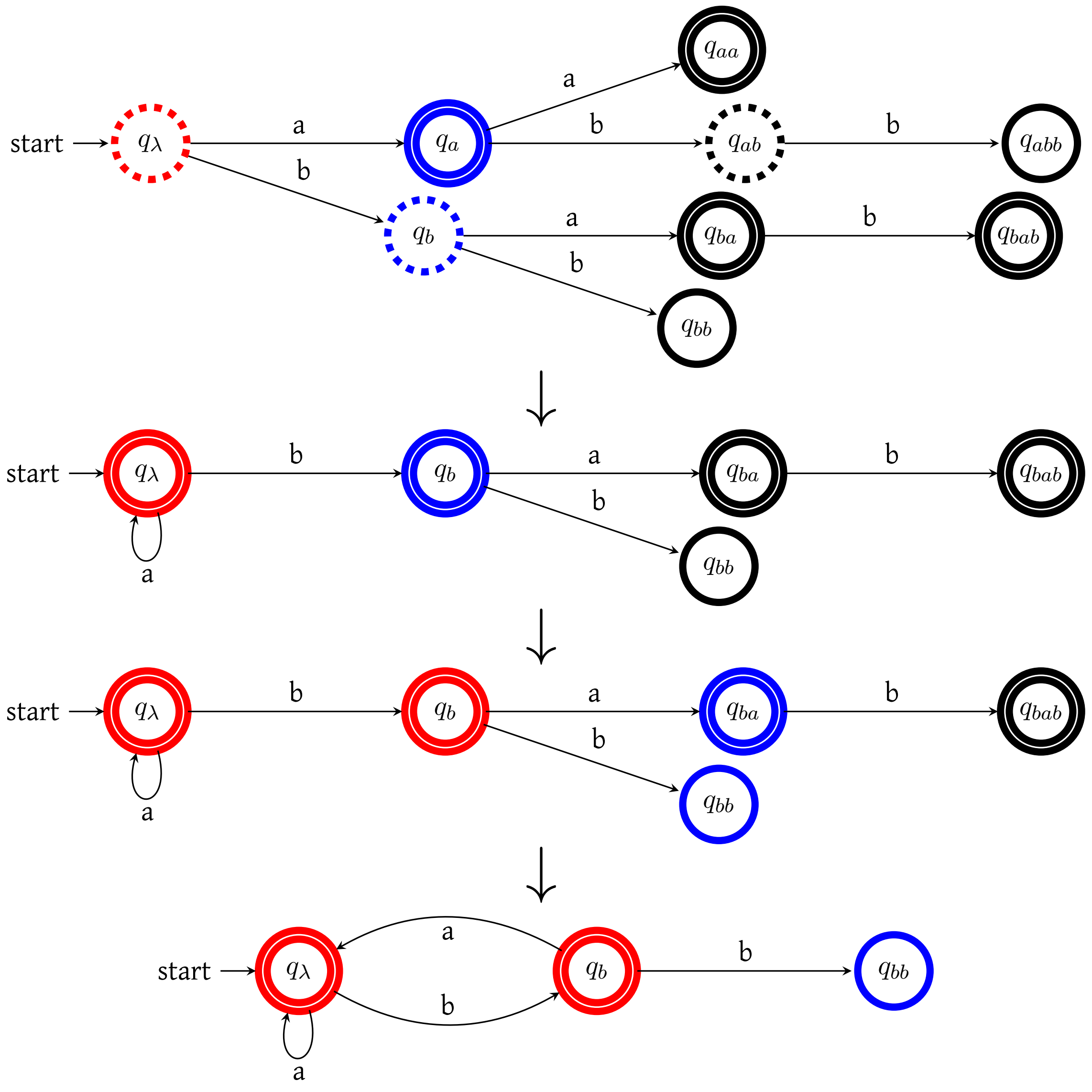


Figure 2.4: Merging of automaton starting with merge $< q_\lambda, q_a >$

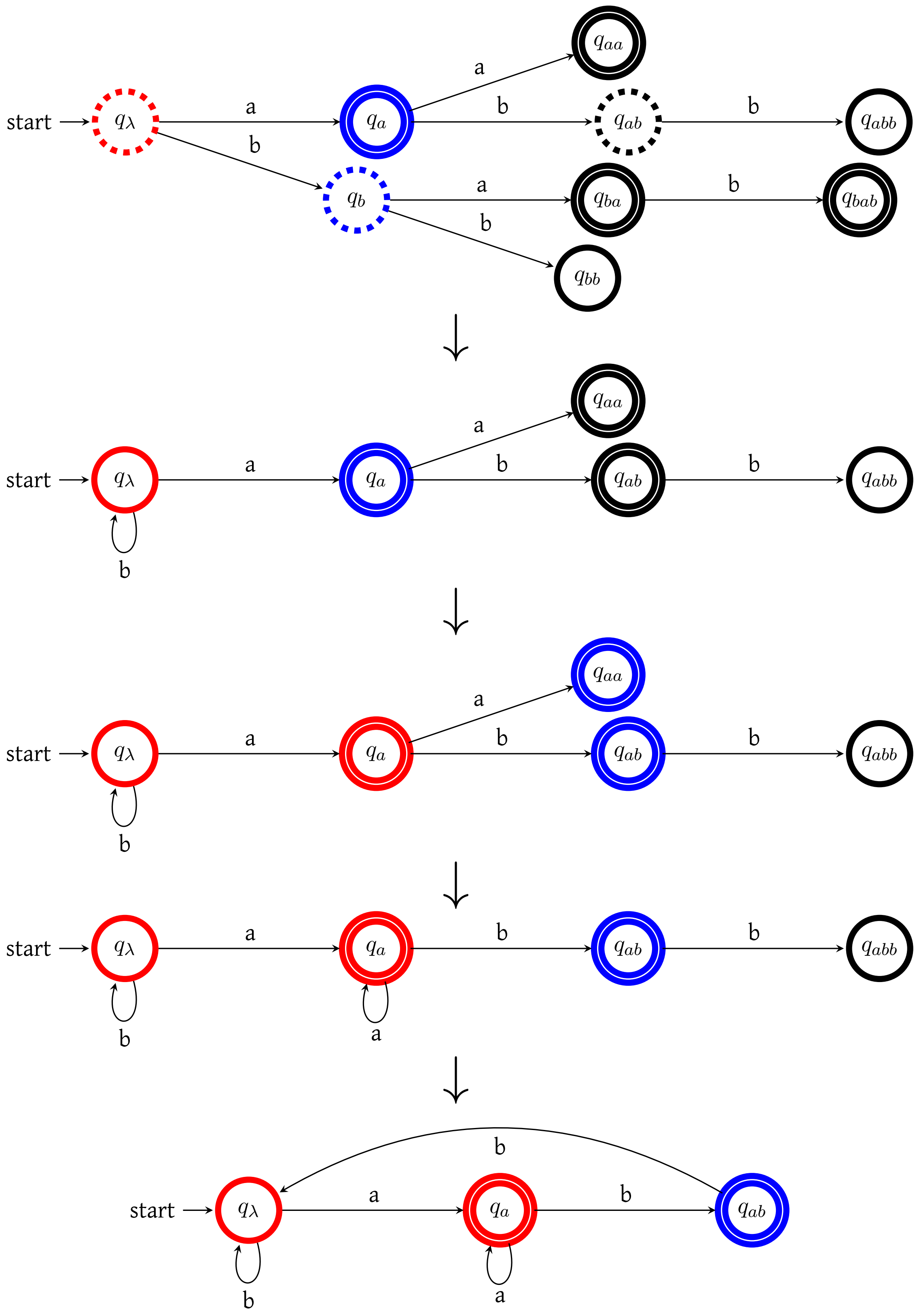


Figure 2.5: Merging of automaton starting with merge $< q, q_b >$

2.3 Databases

2.3.1 Indexes

When accessing data from a database, it is important to realize that accessing data on disk is a lot more time-consuming than data on the (smaller) random-access memory device (RAM). That is why database algorithms must use specialized indexing structures that are different from conventional data structures. These algorithms must mostly be optimized to reduce the number of disk pages they visit. A disk page is a chunk of disk space fetched simultaneously. Such a disk page must contain a high fanout.

High fanout means that an important disk page must contain enough information to quickly find the exact disk page to find the information on. Thus, this disk page has many pointers to other disk pages, which is called a high fanout. With these high fanout disk pages, the total number of disk pages fetched is reduced as much as possible. If using trees as data structures, these trees must then have a very small depth [27].

There are many database index structures available with different specializations. We focus on the use of prefix queries as this is specifically important for learning state machines (see section 3.1.1 on page 38).

B-tree B-tree is the default index of PostGreSQL. It relies on the order of the data. When traversing the tree representing this index, for each node, the next child is selected based on whether the searched key is smaller or larger than the element in the current node. This way, an element can quickly be found, and a query that searches for a specific range can quickly give all elements that fall within the range.

GIN GIN is an inverted index. It matches certain features of the data so that for a feature, you can quickly find what rows have that feature. This way, you can, for example, find all documents that contain a certain word. A GIN index can be particularly fast for searching but is usually very slow when updating the index.

GiST GiST is also an inverted index type. However, this indexing is based on containment. When the tree of the index is traversed, a path is taken based on whether the search query is in some way contained for it. Then, you can index using an R-tree that creates various levels of containment, which is useful for, for example, spatial data. Or use it with an RD-tree, which creates for each node in the tree a containment set, which is useful for, for example, text data[1]. When using the GiSt with an RD-tree implementation, it is important to note that the containment set is hashed and the size of the hash can be controlled by setting the parameter `siglen`[2].

SP-GiST SP-Gist is also a tree, like the B-tree, but allows for specific prefix searches to work efficiently. Unlike the B-tree there are no connections from block to block on the same depth anymore.

[1] https://soylu.org/indexes-in-postgresql-gist/
[2] https://alexklibisz.com/2022/02/18/optimizing-postgres-trigram-search

The adjacency of data is purely based on the clustering it made in the beginning. In that regard, it is similar to the GIST index. However, for SP-GiST this clustering must be non-overlapping. The SP-GIST becomes specifically a fast index when the data can easily be clustered in non-overlapping partitions. For text, this could, for example, be as a radix tree[3].

2.3.2 Text search

Full-text search and trigramming Full-text search often relies on lexemes. This is the tokenization of text into normalized words. This process removes conjugations and casing[4]. For abstract traces, this usually does not make sense. (Although it is probably possible for parts of log traces to create clever conversions to lexemes.)

An alternative for lexemes is trigramming. Trigramming, n-gramming, or q-gramming is not an index but a technique for performing a full-text search. It splits a string into n size parts, which in part can be indexed. These parts then represent that string in an index. Trigrams can be used to fill a GIN and GiST index, which can be used by a prefix search. What is more, trigrams can also help to search for more than only prefix queries; they help to find a piece of string anywhere in the full strings. It is thus not limited to the start of the strings [28]. Lastly, it can even be used to speed up regular expression queries[5]. This work is based on Cho and Rajagopalan [29], which is further improved by Google as explained in this blog[6].

Operator families Note that for PostGreSQL indexes, the index is more like an indexing framework. Together with the operator family, it forms the real index. For example, GIN and GiST can use the operator families from the trigramming extension to be used for text indexing [30]. Installing the trigramming extension can be done using `CREATE EXTENSION pg_trgm`, which makes the `gin_trgm_ops` and `gist_trgm_ops` operator families available for the GIN and GiST indexes.

Collations The collation refers to the way to sort a set of characters. The default most efficient collation is the one provided by `libc`, which relies on the byte order of the character encoding. However, depending on the locale setting, different collations exist that take into account the localized order of characters. For example, how to deal with diacritics.

It is important to note that efficiently searching *through* a B-tree depends on what order is defined on the B-tree. That is why pattern character operations (such as a prefix search) are only allowed on B-tree indices when the B-tree index is built with the currently selected collation. This is done by selecting the `text_pattern_ops` operator family for the B-tree index[7]. This is, however, not needed when the current selected collation is the C collation, but this functionality only recently got added in PostGreSQL 15[8] [30].

[3]https://soylu.org/indexes-in-postgresql-sp-gist/
[4]https://www.postgresql.org/docs/current/textsearch-intro.html
[5]https://wiki.postgresql.org/images/6/6c/Index_support_for_regular_expression_search.pdf
[6]https://swtch.com/~rsc/regexp/regexp4.html
[7]https://www.postgresql.org/docs/current/indexes-opclass.html
[8]https://www.postgresql.org/docs/release/15.6/

2.3.3 Prefix query

Implementation There are multiple ways to make a prefix query in PostgreSQL. There is the `starts_with()` function, the `^@` starts-with operator, and the SQL `LIKE 'a-random-string%'` operation. All of these are equivalent, but because of implementation details, not all of these will correctly utilize the indexes that help to speed up these operations. See here[9] for a discussion on this. It is not fully clear what functionality is supported with what index type. Using `EXPLAIN` can help to determine if an index is used properly in the query search [30].

B-tree vs SP-GiST B-tree and SP-GiST are very similar in terms of our intended usage. However, when retrieving a range of results, a B-tree might have to access multiple adjacent blocks to get the full range, while the SP-GiST contains these results fully to its own children. It is, of course, important to note that the SP-GiST still performs mostly better when the data is not overlapping too much.

Radix tries and the ART index The radix tree that can be implemented with the SP-GiST framework is often used in NoSQL key-value databases. These databases will often rely on the newer radix tree implementation called the ART index [31]. What makes these especially interesting for our application is that they can also run in a distributed setting.

2.3.4 Miscellaneous

Including columns inside an index When creating an index structure in PostGreSQL, non-key columns can be included in the index structure with the `INCLUDE` clause. Now, the values of these columns are directly available on the index structure. This is especially useful for adding the trace outcome directly to the index structure of the trace. For PostGreSQL, this is supported for B-tree, GiST, and SP-GiST, but not for GIN[10]. The values are only available on the leaves and are not available for tree navigation.

The COPY protocol of PostGreSQL The COPY protocol is a protocol that allows results to be written into a file. What makes this especially useful is that programs can hijack this protocol to read results directly when they arrive instead of when everything is done and also cancel when a satisfying result has been reached. They can thus "stream" the data instead. This allows more clever use of queries as you do not necessarily have to wait until they are completely finished. This functionality is only very recently available in libraries that can build on top of PostGreSQL[11][12]. It is important to note that the query optimizer will try to optimize your query depending on the total size it is expected to return, even if you cancel the operation halfway with the `COPY` protocol [30].

[9] https://www.postgresql.org/message-id/flat/232599.1633800229%40sss.pgh.pa.us
[10] https://www.postgresql.org/docs/current/sql-createindex.html
[11] https://web.archive.org/web/20231212092242/https://pqxx.org/development/libpqxx/#2023-01-12-at-last-faster-than-c
[12] https://www.psycopg.org/articles/2020/11/15/psycopg3-copy/

2.4 State of the art

Here, we discuss the state of the art for passive learning (Flexfringe) and the state of the art for active learning ($L^{\#}$ and iMAT).

2.4.1 FlexFringe

The blue-fringe algorithm proved to be a very good heuristic for what merges to consider, but still leaves a lot of room for how to calculate the scores for these merges [23]. Instead of just counting, there are various other methods available. These methods are for example Alergia [32], AIC [33], MDI [34], and many more.

FlexFringe [25, 24] is a state machine learning software framework that tries to be extendable for different scoring heuristics. It is intended to learn a state machine from a fixed set of traces, by implementing the blue-fringe algorithm. It provides an interface for new scoring heuristics. FlexFringe already comes with various score and consistency routines and allows you to easily write your own.

FlexFringe allows for different settings to tweak the blue-fringe algorithm. The order in which promotion is done can be set from the most shallow nodes to the most frequent blue nodes. It can also be configured to perform merges between two blue nodes. Next to that, you can add sink nodes. These are nodes that are not considered during the merging because there is not enough information available for them. You can also set final probabilities, which should be set when the ending of sequences contains information as opposed to learning from traces from a sliding window. Moreover, you can alter settings to consider when consistency should be considered. Lastly, you can alter the mode to learn with a Markovian property or to model the kTails algorithm [35]. Lastly, it contains different modes to learn from batches or streams Finally, it also has a "predict" routine to how other traces are classified with a given model.

Most importantly, Flexfringe is the framework by which the algorithm presented in this thesis is implemented.

2.4.2 More research on L^*

There have been several improvements to the L^* algorithm. There is, for example, research on more advanced counterexample processing [20, 19, 36, 37, 38]. There are also deviations in the algorithm, such as using different kinds of queries [39], learning from statistical queries [40, 41, 42], using adaptive distinguishing sequences [43, 44], or learning infitary languages instead [45]. There are also extensions on the algorithm such as learning for, for example, symbolic state machines [46, 47], extended state machines [48], or non-deterministic state machines [49, 50].

The current existing state of the art is the TTT algorithm that replaces the observation table with tree structures [51]. A more recently published algorithm is $L^{\#}$, which performs its operations directly on an observation tree [52, 53]. The latter is discussed in more detail in the next section, as it forms the basis of the proposed algorithm in this thesis.

Algorithm 3: $L^{\#}$ algorithm

```
$\mathcal{T}$ (observation tree with core $S$ and fringe $F$)
Loop
    foreach $p \in F$ do
        if $p$ is isolated then
            LSHARPCLOSE($\mathcal{T}, p$)
        end
        if $p$ is not identified then
            WEAKCOTRANSIVITY($\mathcal{T}, p$)
        end
    end
    if isolated states in $F$ then
        continue (promote more isolated states)
    end
    $\mathcal{H} \leftarrow$ BUILDHYPOTHESIS($\mathcal{T}$)
    $\sigma \leftarrow$ COUNTEREXAMPLEMILKING($\mathcal{H}$) if $\sigma$ then
        PROCESSCOUNTEREXAMPLE($\mathcal{H}, \sigma$)
        continue (Found counterexample without asking the equivalence oracle)
    end
    $\sigma \leftarrow$ EQUIVALENCEORACLE($\mathcal{H}$)
    if $\sigma$ then
        PROCESSCOUNTEREXAMPLE($\mathcal{H}, \sigma$)
        continue (Did not yet find the solution)
    end
    return (Solution found!)
end
```

$L^{\#}$ Algorithm

$L^{\#}$ is an active learning algorithm that operates directly on the observation tree [52, 53]. It does not use any auxiliary data structures and makes most of its decisions based on the notion of apartness. An outline for the algorithm is found in algorithm 3.

Apartness is defined for two states if there is evidence (called the *witness*) that the two states are not equivalent to each other, and thus apart from each other. This is similar to comparing the different rows in an observation table, but now it is defined on the observation tree instead. Apartness is indicated by stating $p\#q$. Here the states p and q are apart from each other. In mathematical notation, we indicate the witness σ as $\sigma \vdash p\#q$. The witness is also known as a *separating sequence* [54].

In the observation tree, states are divided into three categories. The first category is the *identified* nodes in the observation tree indicated by S. The notion of *identified* means that these nodes are separate states in the resulting state machines. These are the same as the RED rows in the observation table for L^*, or the RED nodes in the Blue-fringe algorithm. The second category is the *frontier* nodes, indicated by F. The frontier nodes are one node further than the identified nodes. These are comparable to the BLUE rows in the observation table, or the BLUE nodes in the Blue-fringe algorithm.

The first step of $L^{\#}$ is to check if there are any *isolated* states in F. An isolated state is a state in F that is apart from all states in S. If an isolated state exists, it is promoted to S, and all its transitions to other states are queried and added to F. This operation is very similar to the closeness check in L^*.

There is also an operation in $L^{\#}$ that is similar to the consistency check in L^*. This is the weak co-transitivity check in $L^{\#}$. Weak co-transitivity states that if two states ($p \# q$) are apart from each other by witness σ and the transitions for σ are known for another state r, then this state r must also be apart from either p or q. Thus, for any state r in F that is not yet identified, we can find two apart states ($p \# q$), and then the witness for these can help to identify state r, as that witness will make r apart from either p or q.

At some point, the hypothesis is made to check for equivalence. For $L^{\#}$, the hypothesis is made when there are no more isolated states in F. All the states in F don't need to be identified. Thus, there might still be weak co-transitivities to be resolved. However, for states that are not yet identified, a *choice* $h : F \to S$ is made to merge these states to a state from S these states are not apart from.

$L^{\#}$ performs another subroutine before sending this hypothesis to the equivalence checker, and that is the operation of *counterexample milking* [55]. Mistakes made from *choice* h are still corrected.

Then, the hypothesis is accepted or not. When rejected the counterexample processing subroutine is started. For L^*, the counterexample subroutine is based on Rivest and Schapire [20] counterexample processing. This routine splits the separating sequence using binary search. It will make a query for each split and recursively call the routine until the apartness relation is within the frontier.

iMAT

Most active learning techniques rely on the concept of MAT as a teacher (see section 2.2.1 on page 24 for MAT), but Moeller et al. [14] present a new framework called iMAT, which loosens up on the concept of MAT. In this framework, the teacher has become an incomplete teacher.

The incomplete teacher is a teacher who cannot answer all queries anymore. The paper makes the following changes to the teacher to be incomplete. First, it changes the membership queries. It loosens the requirement that the teacher should always be able to answer the membership queries. Instead, it can also answer with a "Don't Know". It also loosens the requirement that the teacher should be able to answer equivalence queries. Instead, it adds a new type of query, namely the *distinguish query*: Queries that only answer if two states are the same or not.

In the paper presenting iMAT, Moeller et al. [14] also present an algorithm to learn state machines from iMAT, called LstarBlanks. LstarBlanks is a modified L^* algorithm (see section 2.2.1 on page 25 for L^*) that uses an SMT solver to propose solutions to the observation table with the missing entries. Then, from the solutions called the worklist, the smallest state machine is picked, and the search continues. Picking the smallest solution ensures that the resulting answer is always the smallest state machine.

With the new concept of iMAT, you can learn state machines from a fixed set of traces with an active learning approach. It is as Moeller et al. [14] describes a "bridge between active and passive learning".

Moreover, iMAT is easier to implement in practical settings than MAT. The requirements for MAT can just not always be fulfilled in real-world settings.

There have been other older papers presenting a loosened version of iMAT. In these papers, they called the teachers the inexperienced teachers. For example, Grinchtein and Leucker [56] present a very similar case with an algorithm that also relies on SAT solving. Unfortunately, it misses a description of the exact algorithm and has no source code. The paper Leucker and Neider [57] review related work on inexperienced teachers, such as Grinchtein, Leucker, and Piterman [58] who define weak versions of closed/consistent and Chen et al. [59] who introduce containment queries. All these approaches have the CSP (constraint-satisfaction problem) [35] as a basis for their own approach [57].

CHAPTER 3

Methods

3.1 Database queries

The following database queries are written for PostGreSQL. They use `table` as the table name, `trace` as the trace name, and `res` as the outcome of that trace. They also use variables where `:n` is the variable and `:'n'` is the variable quoted (Since PostGreSQL 9.1). In PostGreSQL `||` indicates the joining of two strings. For all queries, you can limit the amount by canceling the operation in a streaming operation or using the `LIMIT` keyword.

3.1.1 Prefix query

The first proposed query is the prefix query. This query takes a prefix trace and finds all the other traces that start with this trace. The prefix query thus finds all the future responses for the state corresponding to the access trace used in the prefix query.

With the SP-GiST indexing, this query is expected to be fairly fast. That index would put the traces in an index that is very similar way to how a PTA would organize them.

For small access traces, this can return the whole database. We thus have to limit it to a certain n. It will then only return an extension of the prefix of the maximum number of symbols. This query is thus noted as PREFIXQUERY(t, n, k) with t being the access trace, n being the max size of the returned traces, and k the amount of traces to return.

```sql
SELECT * FROM table WHERE trace LIKE :'t' || '%' AND LENGTH(trace) < :n LIMIT :k;
```

Snippet 1: Select k traces that start with t and are shorter than n.

3.1.2 Distinguish query

The second type of query is the distinguishing query, as also explained by Moeller et al. [14]. This query can find for two states, as presented by their access trace, different outcomes corresponding to the same extension. This different outcome then indicates that these states are not equivalent. In $L^{\#}$, this query returns the same as an apartness check would do if it tries to look for a *witness* [52, 53]

With this query, we can implement an equivalence oracle, but it could also aid with state merging. We can also limit this search again up to a certain n. This query is then noted as DISTINGUISHQUERY($t1$, $t2$, $n1$, $n2$) with $t1$ and $t2$ the two different access traces and $n1$ and $n2$ the max size of the returning traces.

```sql
SELECT * FROM (
    SELECT * FROM table WHERE trace LIKE :'t1' || '%' AND LENGTH(trace) < :n1) q1
INNER JOIN (
    SELECT * FROM table WHERE trace LIKE :'t2' || '%' AND LENGTH(trace) < :n2) q2
ON substring(q1.trace FROM (LENGTH(:'t1') + 1)) = substring(q2.trace FROM LENGTH(:'t2') + 1)
WHERE q1.type != q2.type;
```

Snippet 2: With two prefix sub-queries, we join them on equal suffix string and then check if the outcome is different.

3.1.3 Regex query

A state machine can be completely captured within a regular expression (regex) [16]. Such a regular expression describes all the possible edges and paths in a single string. The database can be queried using a regex as well. Given the regex and the outcome of a state of a set of states, the database can check if the traces it holds agree with that outcome.

This query is noted as REGEXQUERY(r, o) with r the regex and o the expected outcome for that regex. Note that the regex must be anchored with "start" (^) and "end" ($) anchors. The performance of the regex query can be increased by using indexes based on trigramming[1].

```sql
SELECT * FROM table WHERE trace ~ '^(' || :'r' || ')$' and res != :o;
```

Snippet 3: Select all queries that match the regex but do not match the outcome.

Regex construction

However, constructing the regex from a DFA is not a trivial task. There exists a plethora of different regex constructing algorithms [16, 60, 61, 62, 63] which all behave different and have different

[1]https://wiki.postgresql.org/images/6/6c/Index_support_for_regular_expression_search.pdf

(equivalent) outcomes from the same problem.

The most intuitive method is the state elimination method [16]. This method removes the states of the state machine in some order and replaces their edges by creating new edges that represent the regex between the states surrounding the state to remove. Figure 3.1 depicts how this would work for a (part) of a state machine. The order of the states to remove can result in a different regex with a smaller size. One of the heuristics that seems to work very well is selecting the state to remove that has the smallest number of connecting edges [64].

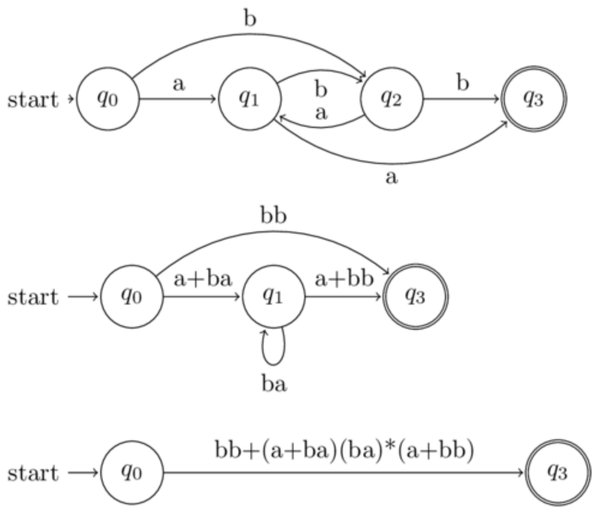


Figure 3.1: Reducing a (part) of a state machine to a regex. Please note that alternation is indicated with a plus (+) instead of the vertical bar (|). This is because it is written with Kleene algebra instead of what is conventionally used for regular expressions [62]. Also note that this is just part of the state machine, as it is unknown where the outgoing edges of $q3$ go to. Finalizing of the regex happens by placing the start (ˆ) and end ($) anchors. Modified picture from StackExchange[2]

3.2 Algorithm: DAALder algorithm

3.2.1 Motivation

The proposed algorithm in this thesis, called DAALder, is inspired by the $L^{\#}$ algorithm (see section 2.4.2). The reasoning for that is as follows. The database as discussed here can be seen as the system under learn (SUL) and a form of the iMAT teacher (see section 25) [14]. However, the SMT solver used by Moeller et al. [14] in the algorithm to learn from iMAT seems to be too heavy to have a performant algorithm. Instead, the following paragraph explains that a modified state-merging algorithm like $L^{\#}$ seems to be the better choice.

$L^{\#}$ performs its search directly with the observation tree. As this observation tree is the same data structure that also underlies the blue-fringe algorithm in the form of an APTA tree, it seems possible to combine active learning and passive learning this way with the $L^{\#}$ algorithm. In this algorithm,

[2]https://cs.stackexchange.com/q/28517/113084

exploration of the new states follows a similar structure as proposed by the $L^{\#}$ algorithm. Creating hypotheses from the emerging observation tree can then be done by state-merging algorithms. Moreoever, this would also allow to extend the state merging scoring heuristics with more data that is present in the database. When data is saved to an observation table, it is not directly evident how extensions to state merging algorithms can be added to the observation table as they are usually designed with trees as datastructures in mind.

Important insights and properties that arise to formulate this algorithm are: (1) the observation tree in $L^{\#}$ is similar to the APTA tree in the red-blue framework, (2) apartness is somewhat the same an consistency check, (3) $L^{\#}$ operates on a partial Mealy machine, just like you would with an iMAT, (4) merge probability in the red-blue framework can be used to guide exploration, and (5) extending existing merge heuristics can help research further to improve this specific algorithm, even including more information present in the database or using different database queries.

3.2.2 DAALder algorithm outline

A sketch of the algorithm is depicted in figure 3.2, and an outline of the pseudocode of the DAALder algorithm is presented in algorithm 4.

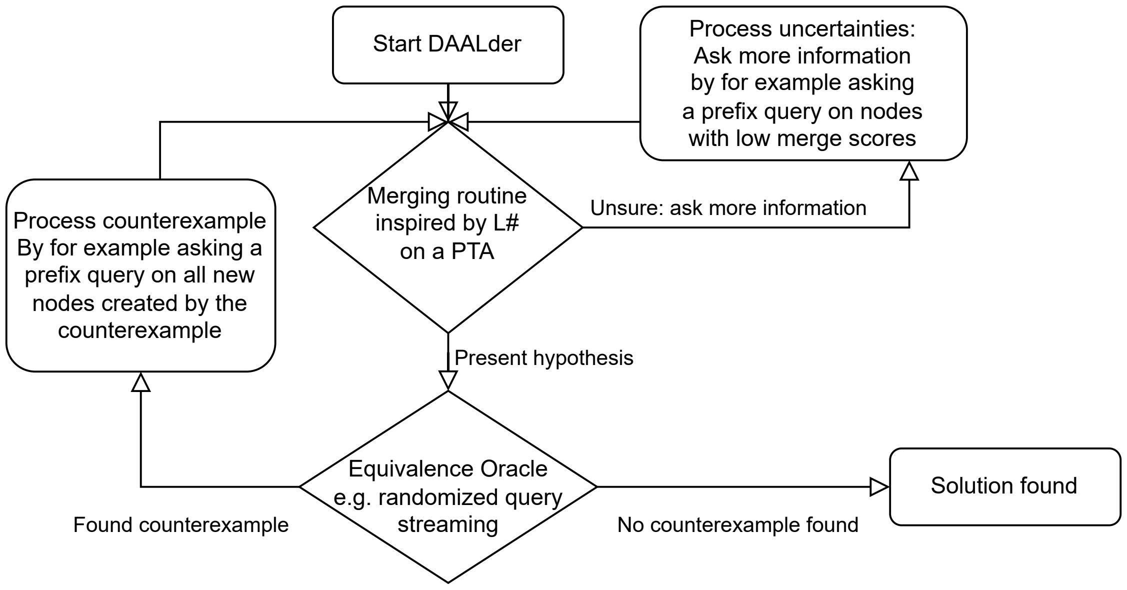

Start DAALder
Process uncertainties: Ask more information by for example asking a prefix query on nodes with low merge scores
Process counterexample By for example asking a prefix query on all new nodes created by the counterexample
Merging routine inspired by L# on a PTA
Unsure: ask more information
Present hypothesis
Equivalence Oracle e.g. randomized query streaming
Found counterexample
No counterexample found
Solution found

Figure 3.2: Graphical representation of DAALder

3.2.3 Differences with $L^{\#}$

Blue node completion

In the original implementation of $L^{\#}$ [52, 53] every time a node was labeled blue, the incomplete transitions were asked as well. For $L^{\#}$, this perfectly makes sense and can, with some alteration, also be done in DAALder. Instead of asking for the next transitions, a prefix query can find n traces that start with the node's access trace. We call this BLUENODECOMPLETION.

Algorithm 4: DAALder algorithm

```
Data: A set of traces with an output label
Result: A hypothesis $\mathcal{H}$ consistent with the provided traces
Initialize: $\mathcal{T}$ a PTA (observation tree) with red core $S$ and blue fringe $F$
Initialize: $S$ red core nodes, initially with a root node
Initialize: $F$ blue fringe nodes, initialize using some random traces
Loop
    state_isolated ← $false$
    unidentified ← $\varnothing$ (The blue states without merge candidate)
    all_merges_todo ← $\varnothing$ (Merges to do, to make the hypothesis state machine)
    foreach $p \in F$ do
        possible_merges ← $0$
        merge_candidate ← $?$
        foreach $q \in S$ do
            if CheckConsistency($p$, $q$) then
                possible_merges ← possible_merges $+ 1$
                merge_candidate ← $q$
            end
        end
        if possible_merges $= 1$ then
            all_merges_todo $add$($p$, merge_candidate) (identified node)
        end
        if possible_merges $= 0$ then
            DAALPromote($p$)
            state_isolated ← $true$
        end
        if possible_merges $> 1$ then
            unidentified $add$ $p$ (Investigate these states more)
        end
    end
    (explore_more, merges_todo) ← DAALProcessUnidentified(unidentified)
    all_merges_todo ← all_merges_todo $\cup$ merges_todo
    if explore_more $or$ state_isolated then
        continue (Investigate the added traces and/or a new fringe/frontier)
    end
    $\mathcal{H}$ ← BuildHypothesis($\mathcal{T}$, all_merges_todo)
    $\sigma$ ← EquivalenceOracle($\mathcal{H}$)
    if $\sigma$ then
        ProcessCounterExample($\mathcal{T}$, $\sigma$)
        continue (Did not found the solution yet)
    end
    return $\mathcal{H}$ (Solution found!)
end
```

DAALProcessUnidentified and ProcessCounterExample add new traces to the observation tree. In our implementation DAALProcessUnidentified asks for more information on nodes when the highest merge score is less than twice the second-largest merge score.

Another advantage we have in FlexFringe, is that there are multiple access traces available for each node. This is because of the union-find structures that quickly give access to all previously merged nodes [24, 25]. This thus allows for a complete list of access traces that all end in this specific node to be completed, except for just one node.

A FlexFringe limitation

There is one big problem with this and with other exploring during merging techniques. FlexFringe is not designed to handle new information when a part of the state machine is already constructed. This messes up the links (if information is added to red nodes) and the statistics (if information is added to blue nodes). It has to be noted that this is not a theoretical limitation, but rather a mere implementation obstacle that would require a significant portion of refactoring in the current implementation. To circumvent this refactoring process, which is out of scope of this work the following steps are taken: Whenever new information should be added, the state machine must be unrolled to be an APTA tree again, and then new information can be added again. With the union-find implementation, this can, however, be quite efficiently done [24, 25], but should still be done in batches for performance reasons.

Other differences

In $L^{\#}$, there are additional checks on consistency when the hypothesis is constructed. This is because in $L^{\#}$, if a blue node is unidentified, it is merged according to some *choice h*. For FlexFringe, a merge is only considered viable if it is indeed consistent. Consistency is thus kept in check throughout all the mergers.

3.2.4 Exploration during merging

Discovering the unidentified portions

A part of the algorithm that is left open is what to do whenever it is unclear where a blue node should be merged. This is the subroutine DAALPROCESSUNIDENTIFIED, which is what to do when a state is unidentified. In the description of the $L^{\#}$ algorithm, it is also left unclear what merge should be made. In the $L^{\#}$ algorithm, it is referred to as *choice h* without any clear description of whether this is, for example, random or not. Here, we explore a few strategies that can be employed.

Staying close to passive learning Instead of doing something somewhat more intricated, we could just take the best merge as given by the scoring of merge heuristics. We call this approach simply JUSTTAKEBESTMERGE.

In fact, all the iteration steps that are done by the DAALder can just be skipped and the best merge by score can always be taken. Then, the algorithm would be alternating between taking the merges associated with the best scores and then asking an equivalence query. We call this approach ALWAYSTAKEBESTMERGE

These approaches do not rely on any additional exploration. These approaches, thus, would rely much more on the equivalence query. Since the equivalence queries are usually quite compute-intensive to execute, it is not one of the best approaches. However, for database-assisted learning, how compute-intensive this is is dependent on the size of the database.

Adding exploration A better approach would be to take a look at how exploration can be built into this subroutine. When a set of merges is considered, merge heuristics can tell which one is most likely to be made. Moreover, it can also tell, given the scores, how certain it is that the best merge is indeed the best. Then, if a merge is the best, but the second-best merge is fairly close, additional action can be taken instead of naively merging.

If there is doubt between a pair of merges, you can ask for more information for these nodes. You could, for example, ask a prefix query for the access traces of these nodes. You could even ask prefix queries for the list of access traces that are represented by that node. This way, you can gather more information that will help you decide which one of the merges should be taken.

In the implementation that is tested, this happens when the factor between the best merge and the second best merge is two. When a merge is indeed the best, it should be fairly obvious what the best merge is. However, this number can be tweaked if it is set too high or too low.

Example For an example of this, let us look back at the example from section 18 on page 28. If we remove aa from the example, it is no longer clear what merge to make. Now, instead of making an arbitrary merge, DAALder asks a PrefixQuery on a and b. Then, when aa is found with this PREFIXQUERY, there is now a clear merge to be made. The intuition behind this exploration is that when it is not clear what merge to make from the merge heuristics, it is likely that after asking for more data starting with one of the access traces of the nodes involved, it will be clear what the correct merge is.

With distinguishing queries Alternatively, given an uncertain merge, it is also possible to ask the database whether that merge could be possible. For two nodes, a distinguishing query on the access traces can be made to check if two nodes can be merged. If this query returns any traces, you can be sure that these nodes must not be merged. To be even more rigorous, you can also check all the combinations of access traces of the two nodes, not just the representative access trace. This additional check could be very compute-intensive, but it is no longer necessary to use an equivalence query.

Equivalence oracles

Checking the equivalence of a state machine is usually the most compute-intensive part of an active learning algorithm. Using a database could make things somewhat easier as large chunks of data can be checked at once. However, it is probably best to call the equivalence oracle as little as possible.

Regular expression A regex query can be used to check equivalence in one query. The regex itself can be constructed to represent the state machine for a specific outcome. Now, you only have to find

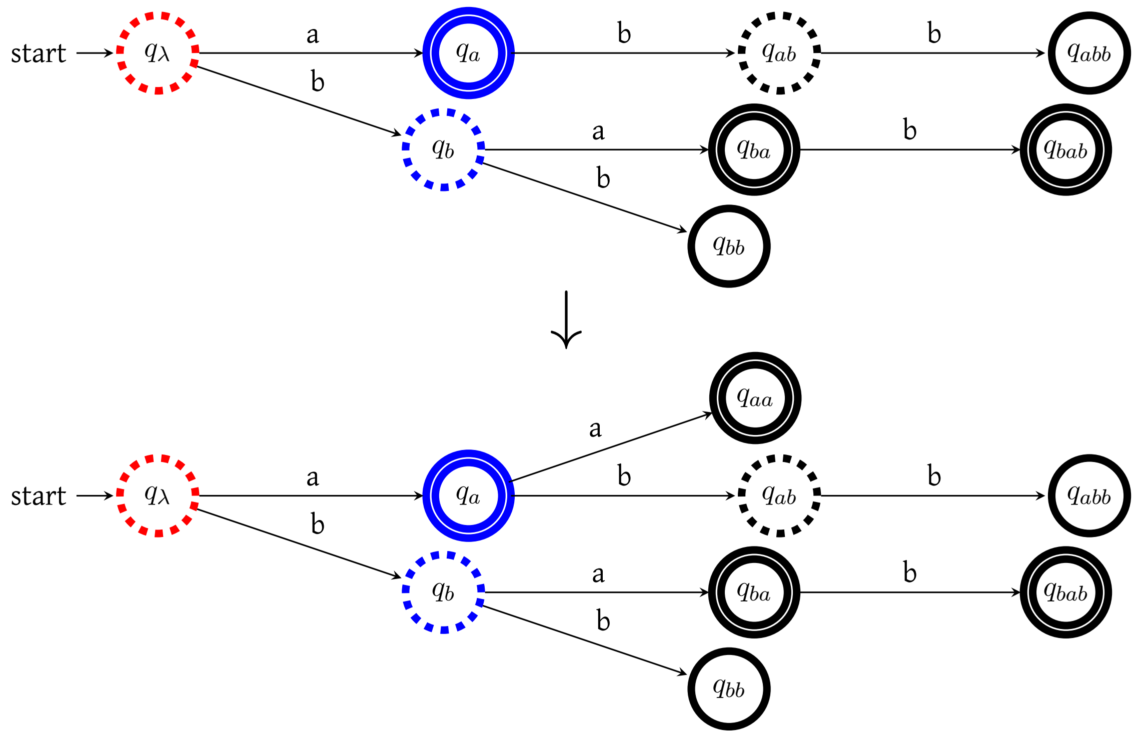


Figure 3.3: In this example, aa was found when asking a PREFIXQUERY on a.

a trace that matches the regex but has a different outcome. That will then be your counterexample. Checking this for the complete set of outcomes completes the equivalence query. For improved searching, the order of outcomes to check could be randomized. Considering all the nodes that are associated with a certain outcome for the regex construction could result in fairly large regexes. Instead, you could review the nodes one by one or consider only a subsection at a time.

Distinguishing queries The equivalence oracle can be built similarly to how it is built in Moeller et al. [14] with distinguishing queries. In FlexFringe, every node contains a list of access traces that are bookmarked and associated with this node during training process. These list of access traces must thus be all equivalent. Calling a distinguishing query for all possible combinations of these access traces must return nothing. If something is returned, this is a counterexample.

The only condition for this to work is that the base of the APTA must consist of a specific set of nodes. These nodes must have two or more access traces bookmarked (because only then a distinguishing query can be called on the two). Then these access traces must have all possible ways a possible trace can start. If these conditions are met, this method can find a counterexample.

To see why this is the case let's start with the most simple case. A simple set of traces that contain all possible ways a trace can start is simple the set of the empty trace λ or ϵ. By definition all traces start with the empty trace. This means that any possible counterexample also starts with the empty trace. Now, whenever a distinguishing query is called on the node that contains an access trace that is the empty trace, the counterexample can be found. By extension, if the empty trace is not involved in any merged nodes, another set that does contain for example all traces with one symbol that covers the complete alphabet is sufficient.

This specific condition makes a bit hard to verify the correctness of this method. In general the empty trace is often involved making this a possible method. But if this is not the case extracing the set of all possible prefixes is not trivial to find. It should also be noted that this set of possible prefixes is smaller than the theoretical set of possible prefixes, because a specific prefix is just not present in the particular dataset that is learned from.

Randomized queries Lastly, a method that provides the most reliable bound on runtime is checking a randomly selected trace until a counterexample is found. To prevent checking a trace multiple times, it should be tracked what traces have already been checked. It should be noted that every time the oracle is requested again with a different hypothesis, this set of checked traces must be reset, since a new hypothesis could have changed previously correct traces that are now falsely categorized. The only traces that always can be excluded are the ones that are used during the construction of the hypothesis.

3.3 Experiments

3.3.1 Test data

Stamina

First, there is the stamina dataset [26]. The stamina set, unfortunately, does not come with solutions for its test set. Even an inquiry over email to the corresponding author did not yield us the solution to the test set. Therefore, the train set has been split into a train and a test set. This split is the same as in the paper by Dieck and Verwer [65] and can be found at Github[3].

FSM-learning generator

The second test is a generated test that used the code from Giantamidis, Tripakis, and Basagiannis [13], which can be found at Github[4]. For additions such as a fix on Linux, a converter for Abbadingo, and a variation with non-uniform sampling, see my fork[5].

This generates Moore traces (traces where the output of every step is known), not Abbadingo traces. For the conversion to Abbadingo traces, a small conversion script is written that takes a few random steps from the Moore trace and creates the Abbadingo sequence from it. It is made sure no duplicates are produced.

The generator, by default, tries to visit every node as uniformly as possible. We also wanted to test non-uniform distributions, so we changed the code to create a weighted state machine with a corresponding sampler. This is found in the `no-uniform-sampling` branch.

[3] https://github.com/SimonDieck/Syntactic_Monoid_Passive_Learner/blob/-/data/staminadata
[4] https://github.com/ggiorikas/FSM-learning
[5] https://github.com/hwalinga/FSM-learning

3.4 Implementation

The implementation of the algorithms is done in C++ and is an extension of FlexFringe [25, 24]. The FlexFringe code is available on Github[6]. My extension is written in the active learning framework written by Robert Baumgartner who based his work on the improvements on FlexFringe from Tom Catshoek. Currently, this is not publicly available in the FlexFringe repository, but this should be in the future. If this is still the case and you look for my code, send me an email at `hielkewalinga@gmail.com`.

[6] https://github.com/tudelft-cda-lab/FlexFringe

CHAPTER 4

Results and discussion

4.1 Failures

Some experiments were failures and were not further expanded on. I summarize these failures here for good reference.

Stamina

When testing on Stamina, it appeared to be very hard to learn an equivalent state machine from the data. The data was too demanding to solve to really test the performance of the algorithm, and it was also too little to see where the new algorithm would really shine.

Regex

When using regex for performing the equivalence query, the program quickly failed on a too-large regex for PostGreSQL. This happened around 20 states.

Regex is known to cause exponential blow-up [66], and even with the heuristics that should mitigate this [64]. What's more, even when choosing only a single end node for the regex, instead of all nodes with the correct label, the regex still turns out way too large.

One data source

For a subset of experiments, a new dataset was generated at each size step. Initially, a gigantic dataset was made at first and then sampled for smaller sets. This resulted in very long runtimes for DAALder, and a lot of experiments did not create a consistent state machine.

Possibly, this is a side effect of the sampler not outputting a trace that has already been output. As a result smaller traces will be output first, and more complex traces come out last. When sampling, these smaller traces are much less likely to end up in the dataset used, and are mixed more with the complex ones. This results in a much harder dataset to learn from. That is why for every datapoint in the comparison between algorithms a new dataset has been generated.

4.2 Database performance

Here I test some of the queries with some different index types. This was one of the first tests done. These tests were run on my Elitebook 8470p laptop. The hardware for this PC is an Intel Core i5-3320M, Samsung SSD 870, and two 8GB Kingston memory sticks. The performaance is a memory speed of 5000 MiB/s, SDD random read speed of 74.35 MiB/s, and a CPU clock speed of 3 Ghz (measured). These tests are run with postgresql 15.6, libpqxx 7.9.0, and clang 16.0.6 on Debian 12.

4.2.1 Index types performance

To test the performance of the index types, I create a test database. In this database there are 2174725 entries with a size of 36, in total 90MB. The size of the alphabet is 4. I create the indices B-tree with the text_pattern_ops, the SP-GiST index, and the gin and gist indices with the trigram extension. In figure 4.1, we can see the time it took to create the index and its size.

Next, I test the time it took the different queries I have devised using the different index types. For testing, I make a separation between three different types of testing. One is the cold cache, which means that the database accesses data for the first time. There is now no caching available. The

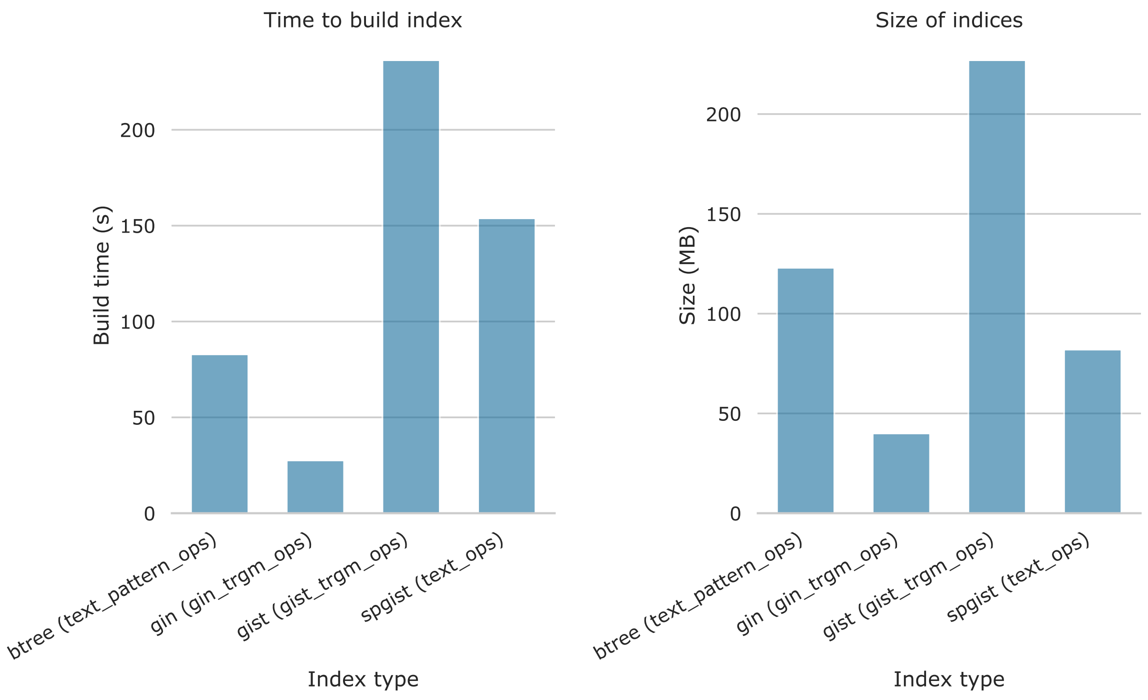


Figure 4.1: Size of the index and duration of building it

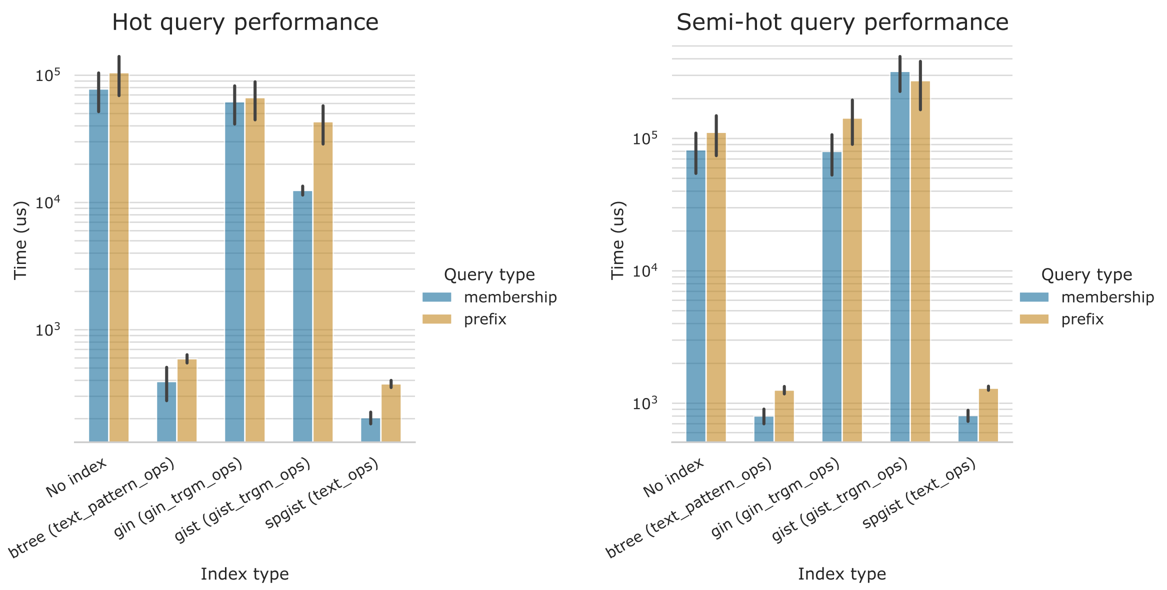


Figure 4.2: Timings of different queries for hot and semi-hot with different index types, error bars are standard error

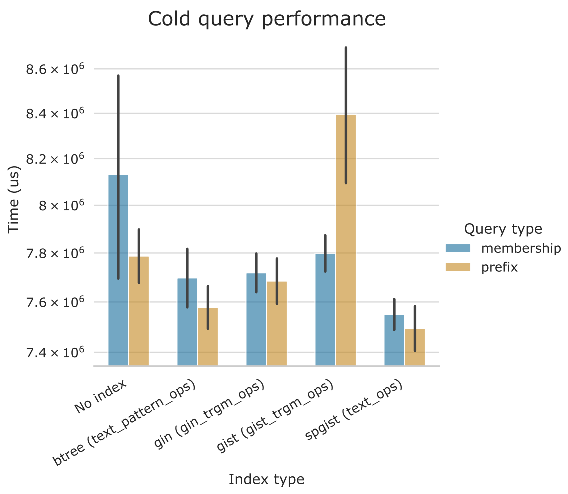


Figure 4.3: Timings of different queries for cold with different index types, error bars are standard error

second type is the hot cache, which means that the data was very recently accessed and is now in the cache. I call an in-between type the semi-hot query, meaning that the exact data is not in the cache but its index is. The test performed by the cold query is done by shutting down PostGreSQL and emptying the cache (send 3 to)`/proc/sys/vm/drop_caches`. The hot query is performed by first running the same query five times. The same is done for the semi-hot query, but different traces are used each time to make it semi-hot. The result can be found in figure 4.2 and 4.3.

Discussion

The B-tree and the SP-GiST index perform very similarly. It was expected that the SP-GiST would perform better, but perhaps because of the small alphabet size, too many blocks had a very large overlap. The same problem of a too-small alphabet was also likely why the trigram-based indices performed so much worse.

4.3 Performance of algorithms

4.3.1 Comparing algorithms

Results

The algorithms that are compared are an implementation of EDSM, a modified version of an iMAT algorithm, and of course DAALder itself. For EDSM default settings from `ini/edsm.ini` from Flexfringe itself are used (from commit `f5d4175`. The modified version of the iMAT algorithm is a variant where the missing values are seen as wildcards and ignored during closeness and consistency checks. This is sometimes also called weak closeness and weak consistency [57]. Finally, to get hypotheses from the observation table with gaps the state merging routine of EDSM is used. The iMAT implementation uses the same counterexample oracle as DAALder. For DAALder the hyperparameter k is set to 10, 100, and 1000. This will help to compare the explore-exploitation trade-off.

The data consists of 17 datasets constructed from 17 different state machines. For each data point a single dataset is used. This data is sampled from randomly generated state machines. The state machine is different for each dataset. The size of the datasets is exponentially increasing, starting with 625 traces and then doubling each time all the way to 40960000 traces. Code to create the state machines and the sampler can be found at Github.[1].

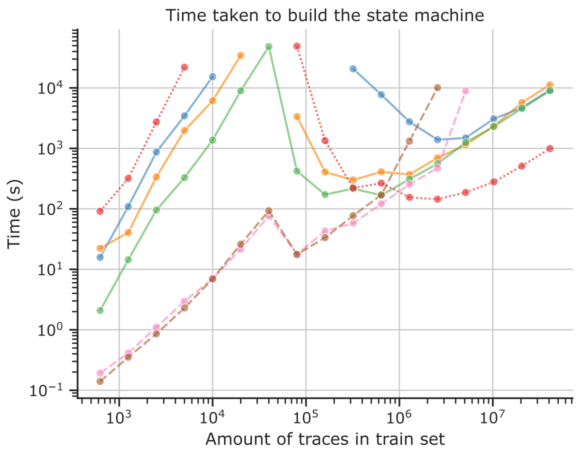
On page 52, the figures for the results on the uniform data can be found, and on page 53, the results on the non-uniformly sampled data can be found.

Whenever a data point is missing, the experiment was stopped. This is because, in the case of DAALder, it took too long, at around 24 hours. Or for EDSM, there was no more memory available of the in total 100 GB, including the 30GB swap memory.

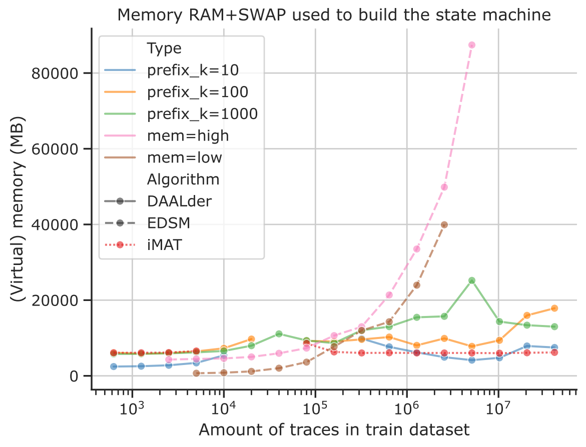
Except for the low memory run (brown line), all the tests are run on a Dell PowerEdge R710 with a Intel(R) Xeon(R) CPU E5620, Seagate ST3146356SS, and Hynix HMT151R7BFR4C-H9 memory sticks, in total 70GB. The performance of the memory speed was measured with `sysbench` as well as the hard disk read speed with mode `rndrd`. The CPU speed was measured under `stress` from `/proc/cpuinfo`. The performance is a memory speed of 4100 MiB/s, HDD random read speed of 2.7 MiB/s, and a CPU speed of 2527 MHz. The computer ran with Ubuntu 22 and PostGreSQL 14.11, libpqxx 7.9.0, and clang 16.0.6.

[1] `https://github.com/hwalinga/FSM-learning`

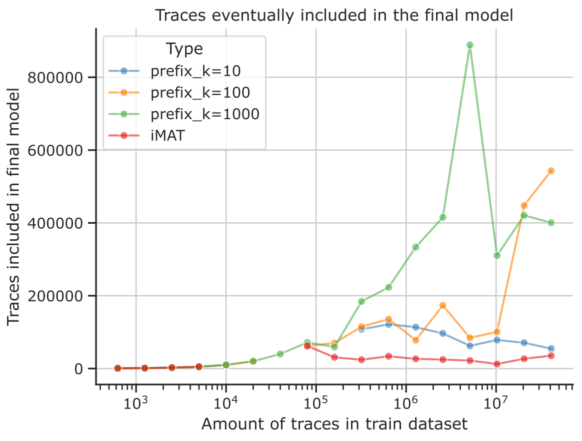
(a) Time vs. size



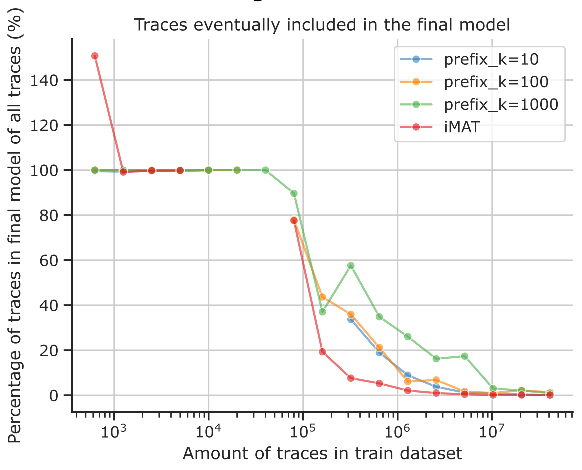
(b) Memory vs. size



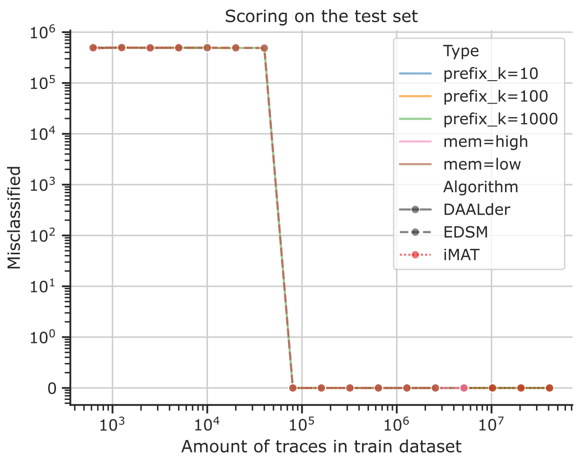
(c) Traces vs. size



(d) Percentage of traces vs. size



(e) Accuracy vs. size



(f) States vs. size

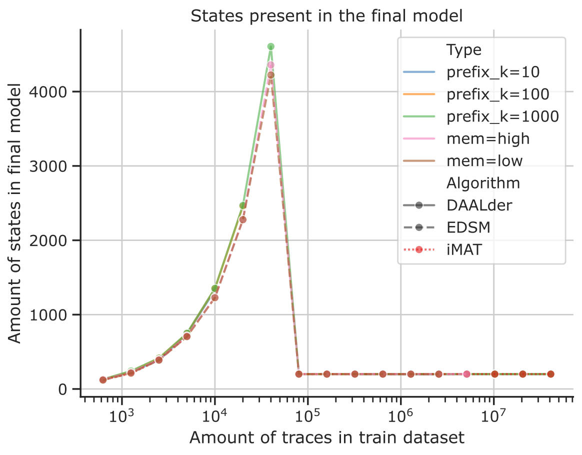
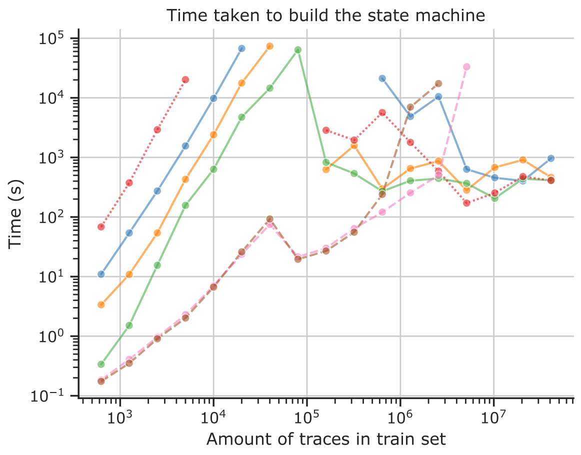
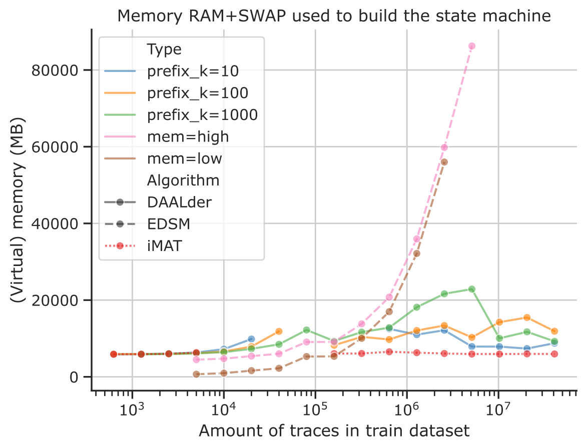


Figure 4.4: Results for data with a uniform sampling approach

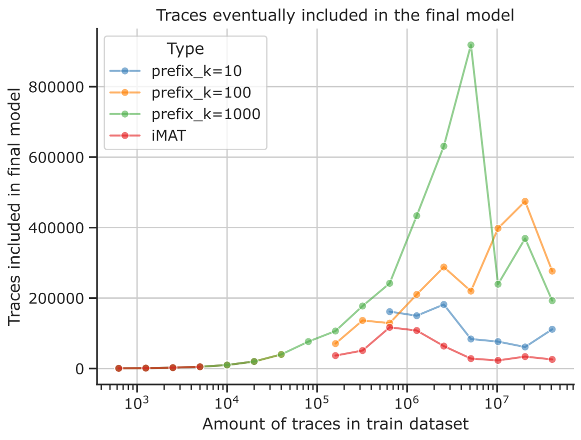
(a) Time vs. size



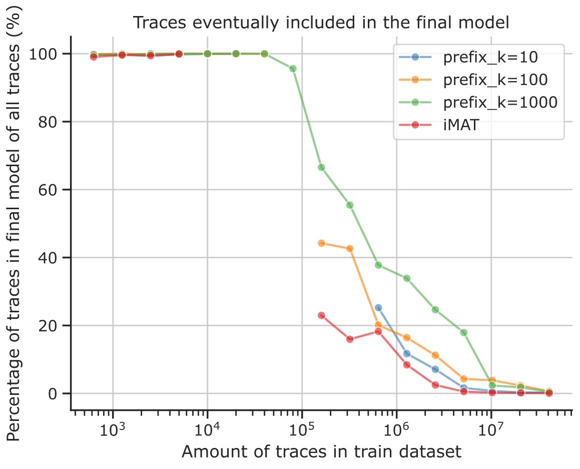
(b) Memory vs. size



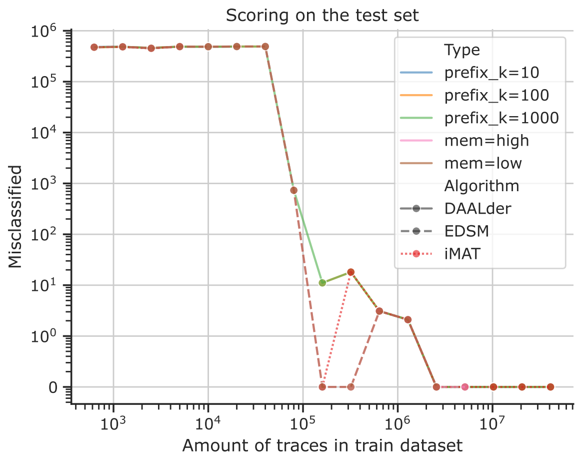
(c) Traces vs. size



(d) Percentage of traces vs. size



(e) Accuracy vs. size



(f) States vs. size

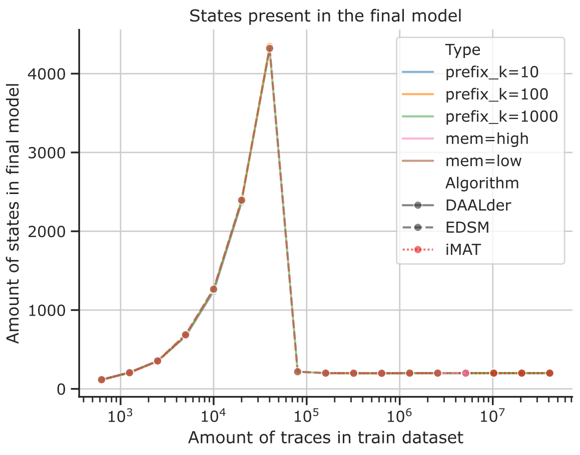


Figure 4.5: Results for data with a non-uniform sampling approach

Discussion

Solvability point When examining the figures there is a clear turning point in the data. This is the point from 40k to 80k traces. Before this point, the algorithms are unable to find the correct state machine as seen in figures 4.4e and 4.5e. After this point, the runs on the uniform data sets find the correct state machine, and for the non-uniform data, only a few traces are misclassified. The 80k seems to be the amount of data that is the solvability tipping point for the algorithms on these kinds of datasets.

After the solvability point, the runtime of the EDSM algorithm drops briefly, as can be seen in figures 4.4a and 4.5a. Before the solvability point, EDSM makes wrong merges which result in a state explosion, as can be seen in figures 4.4f and 4.5f. This state explosion makes EDSM take slightly longer to process the datasets before the solvability point.

For DAALder and iMAT the datasets around the solvability point take the longest to process. Around these points these algorithms try to include every trace one by one, instead of only a subset of traces. It takes a long time to find all the traces in the exploring manner of these algorithms.

Runtime comparison The runtimes vary widely for the different algorithms as can be seen in figures 4.4a and 4.5a. For small datasets, EDSM is much faster than the active learning algorithms. This makes sense as EDSM loads everything in memory and there is not so much to load in memory. When the data grows too big to load in memory, the runtime of EDSM suffers greatly. This is especially true for when the RAM memory is completely used and EDSM resorts to swapping as can be seen in the final data point for EDSM.

The active learning algorithms have a hard time during the solvability point and the datasets leading up to that. After the datasets are large enough to extract a state machine, the runtimes for the active learning algorithms become much shorter. This can be explained by the fact that the active learning algorithms now do not have to include every trace in their model. This can be seen in figures 4.4c and 4.5c for the total traces included and figures 4.4d and 4.5d for the percentage of all traces.

While for the uniform sampled data, the runtimes for the active learning based algorithms continue to grow linearly, this does not happen for the non-uniformly sampled data. Instead, as can be seen in figure 4.5a the runtimes seem to stay the same. It is not fully clear to me why this is the case. I think it might be that it is more likely that the traces that explain some unclear places in the tree can be much less frequent, while for uniformly sampled data, the frequency was always similar.

A characteristic set A characteristic set [18], or tell-tale set [67] of traces is a subset of traces that are sufficient to extract the corresponding state from that can explain the full set of traces. When running the active learning based algorithms, in essence, a characteristic subset is the one that is found. For DAALder on $k = 10$ and iMAT, the size of this subset is relatively constant, as can be seen in 4.4c and 4.5c. For iMAT, this is around 20k-60k traces, and for DAALder, around 55k-160k. This is indeed a bit less than the solvability point, the point where the data starts to contain a characteristic set of traces.

Memory usage Memory usage is depicted in figures 4.4b and 4.5b. Memory usage for EDSM quickly explodes exponentially. For DAALder the memory usage is in the beginning still higher than EDSM, but remains somewhat constant for the larger datasets. The iMAT algorithm uses the least amount of memory. Implementing observation tables can happen in vectorized memory, which is much more memory efficient than maintaining a tree with pointers.

Effect of k A larger number for k makes the DAALder algorithm include more traces more aggressively. This results in a shorter runtime while using somewhat more memory. However, when the datasets become much larger, the effect of k diminishes. The fact that the sparseness drops is likely to make the runs with a smaller k find the correct traces just as fast as with a large k. In fact for the uniformly sampled data, the runtimes are almost the same again for different values of k.

Accuracy comparison For accuracy the amount of misclassified traces is summed. For the data points below the solvability point the accuracy is as good as guessing, meaning that roughly half of the traces were misclassified. After the solvability point, for the uniformly sampled data, the accuracy becomes directly 100% as can be seen in figure 4.4e. For the non-uniformly sampled data, this did not happen immediately as seen in figure 4.5e. For these datasets, a few traces remained wrongly classified until after 2.5M traces were in the train set. What is remarkable is that EDSM performed slightly better in this regard than the other algorithms. This makes sense as EDSM considers all the data at once.

4.4 Miscellaneous

4.4.1 Data loading

The running of the DAALder algorithm did not include loading the data into the database. This would increase the runtime much, as writing to disk and building the index would take a very long time. It is assumed that when these algorithms are used, the data will already be saved in a database. The runtime for loading the data on the database is depicted in figure 4.6.

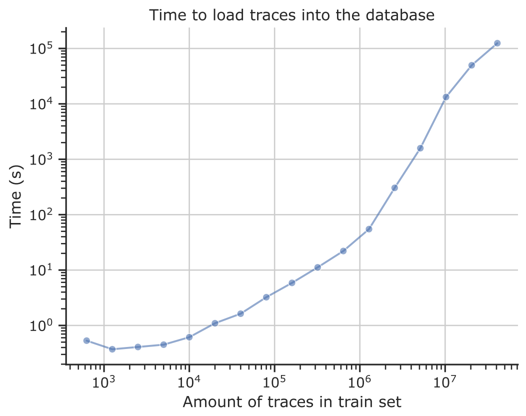


Figure 4.6: Loading times for loading trace data into a database

4.4.2 Does sampling work?

Instead of searching for the correct traces, it might also be possible to just select a random set of traces from a very large set of traces and learn from this. If this set is large enough there is a good chance it is sufficient to learn the equivalent state machine. For this research a trace data set with the size of 20.48M traces is sampled. The result are depicted in figure 4.7.

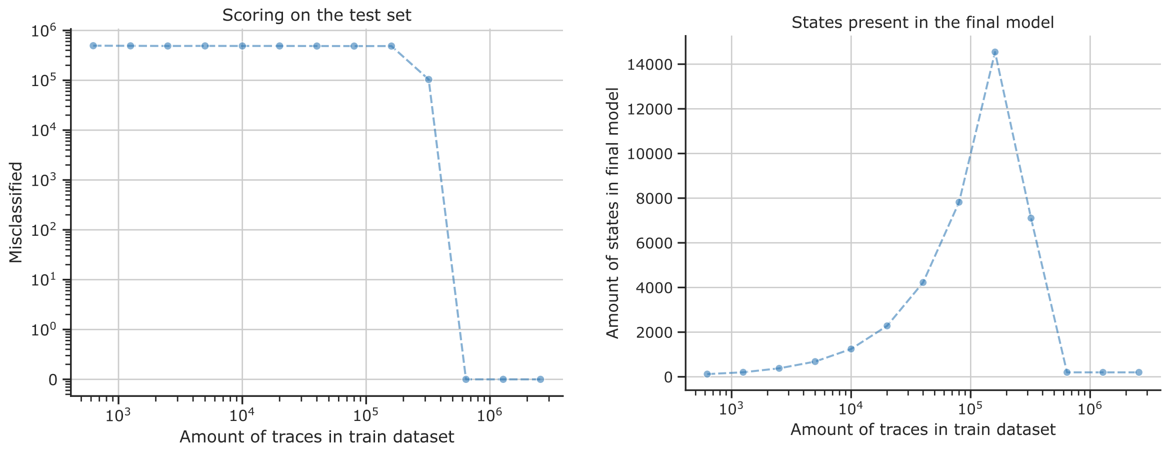


Figure 4.7: Accuracy and amount of states when learning from sampled data.

Sampling seems to be a viable method as well to deal with larger data. The size of the data required is still pretty large. At around 640k traces were required to learn the equivalent state machine. This is ten fold larger than the previously investigated datasets. It is, however, still in the range where EDSM is (slightly) more performant than DAALder. It should be noted that for real life datasets, it could be that a single trace covers a particular important path in the state machine. There is a high chance the sampling approach would miss this.

4.5 When to pick DAALder

For iMAT, the runtimes do not drop directly on the non-uniformly sampled datasets. While for the uniformly sampled datasets, iMAT quickly becomes faster than DAALder again, this is not the case for the non-uniformly sampled datasets. Here, iMAT performs on par with the DAALder algorithm. This is likely because the sparseness of the dataset becomes a problem for iMAT. This results in too many empty spots in the observation table, which slows down iMAT.

DAALder would thus only be an appropriate algorithm for this specific case. On non-uniformly datasets when the data is too large for EDSM but still sparse enough that an algorithm such as iMAT performs worse. Ideally the data is also already in a database present.

CHAPTER 5

Future work

Since this is more explorative research in a relatively new field, there is still plenty more work to do. This is a list of ideas that came to me during my brainstorming sessions with Sicco Verwer and Robert Baumgartner.

5.1 Braoder comparison with different algorithms and datasets

5.1.1 Different algorithms

The current research still misses comparison with different algorithms. For example, FlexFringe also comes with a stream module and a batch module. These modules have the same goal as my research, but this research misses a comparison with the performance of these modules.

5.1.2 Different datasets

The current research also misses an analysis of the performance of the algorithm on different datasets. For example, how the algorithms would perform when the alphabet is much larger. It is expected that algorithms based on tree algorithms can better deal with larger input algorithms than the algorithms that rely on tables. Trees are better at saving data with this sparsity, while tables have to reserve spots for all extensions, which can get really big with bigger input alphabets. Also, it would be interesting to see how the algorithms perform when the corresponding state machine is more sparse. In that case, tree algorithms would also be expected to perform better. If a future reader is interested in how these algorithms deal with larger output alphabets. In that case, it must be noted that FlexFringe's current implementation of EDSM is hardcoded for an output alphabet of two.

5.2 Implementation improvements

5.2.1 Improved regex building

Different regex constructing algorithms

When converting a DFA to a regular expression (regex), there are many ways to get an equivalent regular expression. There has been some research into what the best method for finding this regular expression is [64]. For this project, the most simple intuitive way with a simple heuristic has been implemented. This is the state elimination algorithm where the least connected node at each iteration is selected for removal. However, it could be that implementing the subautomaton decomposition that is mentioned in that paper could yield even smaller regexes.

There also exists another regex algorithm next to the implemented node elimination algorithm. These are the Brzozowski and McCluskey algorithm [60], Arden's method [61], Kleene's algorithms [62] and the McNaughton and Yamada algorithm [63]. One thing all these methods have in common, however, is that there is a chosen order of states to eliminate. What's more is that the chosen order of which states to eliminate first, will result in a different regex with a different size. It could be that depending on what kind of state machine you are dealing with different algorithms result in a regex with a smaller size. Finding the smallest regex is, unfortunately, a PSPACE-hard problem. Although that statement seems to be part of the folklore of Automata Theory[1].

Selecting the most important parts

Instead of constructing a regex for the complete DFA, a smaller regex can be achieved by making only a regex for a part of the automaton. One improvement has already been made, where a selection of the final states is chosen to construct a regex. However, you might be able to construct even smaller regexes by removing some of the unimportant edges from the DFA. This would help to find only counterexamples that show where the core of the current state machine is wrong. Alternatively, it might be better to keep some of the uncertain edges instead, so that you can find counterexamples that disprove those uncertain edges.

5.2.2 Being able to add traces to a partial mealy machine

One of the limitations of FlexFringe is that it cannot add more traces to a partially constructed state machine (as described in section 40). The merges must be undone before new data can be added. This is just not implemented for FlexFringe. With this implemented, you can learn state machines with FlexFringe actually incrementally growing the state machine. However, this is not the most engaging thing to do and is, at the moment, not a real bottleneck as the find-union structure of the APTA in FlexFringe can be very simply unrolled. Although, in terms of runtime, this can be a significant improvement.

[1]https://cstheory.stackexchange.com/a/3983

5.3 Extensions

5.3.1 Investigation into more query types

This thesis proposed a few different kinds of queries that could be used on the database to learn state machines, but there might be other queries we can implement on such a database. Maybe we are able to a subset query [39] or implement a form of a statistical query [42, 41, 40]. Lastly, the regex query is not extensively tested, and the distinguishing query is only partially implemented.

5.3.2 Investigation into the search algorithm

The DAALder algorithm proposed in this thesis is just one approach of many. There might be more approaches, as well as using different heuristics within DAALder to better guide the explore-exploit trade-off. Also, when you take into account that there are different queries could be possible.

5.3.3 Other forms of the state machine

Next to simple Moore machines, there are different forms of state machines, such as PDFA, but also extended state machines [48] and symbolic state machines [46]. It would be interesting to see if the algorithm can be adapted to learn these kinds of state machines as well.

Using the guards of FlexFringe

For extended state machines, FlexFringe already keeps track of guards in the core of its state merging algorithm. It should thus not be so hard to rewire data from the database towards places where the guard logic of FlexFringe can be applied.

Adding statistics to the database

Merge heuristics can be aided by using statistics for what traces come next. Based on these statistics, the algorithm can then consider if a merge should be taken. PDFAs can only reliably be constructed with these statistics present.

These statistics can be calculated on the fly, but that would require the database to do these counts very frequently. However, it would be better if these statistics were cached. In such an implementation these statistics are available for every access trace. To improve the calculation, these statistics can first be added from the leaves, and subsequent calculations can then use the already existing counts from the downstream nodes. To decrease disk size, you could make these statistics only available at a selected range of depths (with an increased frequency at the base of the tree), and getting these statistics by performing a join on the table that holds these statistics.

5.3.4 Incorporating data integration techniques

In a real-world setting, there will be a lot of information available for all the traces, perhaps even scattered among different data sources [68]. The trick will be to find out how to combine these different data sources to find enough support to block or pass a potential merge. Data integration techniques could help with combining the correct information [69].

5.4 Different implementations

5.4.1 Using another database more tailored towards the problem

Different database implementations may improve the algorithm's performance and open up more possibilities for designing new types of queries. In addition to the standard relational databases, there are also ones specialized for text search, most often built on top of the Apache Lucene framework. Text search specialized databases are ElasticDB (and its fork OpenSearch), CrateDB, Solr, or Sphinx, which can cooperate with PostgreSQL, MariaDB (shipped by default), and MySQL. Moreover, document-based databases, such as MongoDB, or key-value stores, such as ScyllaDB DynomaDB, or RocksDB, are very popular and scale extremely well on different machines since sharding is a lot easier to implement for document-based databases. What's more, some databases are designed for graph structures, such as NitrosBase, Blazegraph, Ontotext, Neptune, Virtuoso (with SPARQL), Nabulagraph, InfinityGraph, OrientDB, Giraph (with Java), ArongaDB (also has support for text search, based on RocksDB), Neo4j (with Cypher), and OrangaDB. There are also some graph databases based on key-value store RocksDB, which are Oxigraph and ArongaDB uses a precise vector space model for its search engine and also supports text search directly. Finally, one could hack something with XPath on BaseX for XML searching. Selecting one of these databases would open a lot more possibilities for building your own data structures on these databases.

5.4.2 RIIR

I am just joking. I think the codebase is perfectly fine in C++. However, you could probably find someone eager to do this, and create more appeal for others to work on it.

5.4.3 Circumventing implementation details altogether using Datalog

When reflecting back on what this research actually did, the research seemed to boil down mostly on the rewiring of how data is organized for the evidence-driven merging algorithm. Some alteration is made to limit the amount of data that is examined, but in essence, it is mostly the rewiring of how the data creates a minimal state machine. Take a look, for example, at how the data is indexed inside a PostGreSQL database. It is already indexed using graphs. In fact, it should be possible to use these graph structures on disk directly instead of only querying on them.

One of the big reasons this is a problem is because the graph data structure is a difficult data structure. There are many ways to implement a graph, and its implementation has a direct effect on the performance of the algorithm that uses it. An efficient implementation will differ depending on the graph's sparsity or size. One of the issues is that for a graph a relation must be implemented. A natural way to implement a relation is using a pointer. However, vectorized algorithms will be more cache-friendly and thus much faster than pointer following.

This was all outlined in an extensive blog[2] post of Hillel Wayne (recommended!). The conclusion is somewhat grim: You cannot have a universal graph implementation. However, a few days later somewhat claimed that "the missing datatype" does indeed exist[3]. It has only been in development since the 70's. The proposed framework you should seek this solution in is Datalog, a variant of Prolog.

The idea of Datalog is that the graph algorithm is written in a declarative language (Datalog) and the exact execution is left over to a Datalog execution engine. This engine can then decide what the optimal execution is given statistics on the data. What is more, there even exists a way to make the execution run on the GPU.

The idea is thus to write an evidence-based driven state merging algorithm on Datalog and by selecting a correct execution engine, get most improvements I tried to do in this thesis, just automatically. The only problem is in the implementation: Prolog is not taught anymore on the TU Delft, and improving just the implementation holds less scientific value.

[2] https://hillelwayne.com/post/graph-types/
[3] https://tylerhou.com/posts/datalog-go-brrr/

CHAPTER 6

Conclusion

6.1 Research questions

6.1.1 RQ1: What kind of queries can we ask a database to learn state machines?

Next to normal membership queries, we can ask different queries as well. We can ask prefix queries, which can also be supported by database indexes. We can ask distinguishing queries, relying on the same kind of indexes. Lastly, a regex query can be used as an equivalence query. However, for regex queries, the current state machine has to be significantly small for the regex to be manageable. That is why for an equivalence query, distinguishing queries, or just random membership queries can be used as well.

6.1.2 RQ2: What kind of algorithm can we use to learn a state machine from a database?

We can learn from the database using a combination of active learning and passive learning. The algorithm should contain a part that selects what queries to ask from the database (active learning) as well as a part that creates a state machine from this data. One such algorithm is the DAALder algorithm. Some variations can be made to this algorithm as well. It has parts, such as the DAALPROCESSUNIDENTIFIED subroutine, that can take on various forms. This part must take into account an explore-exploitation trade-off. Creating an equivalence query remains a challenge, but it can be circumvented by leaning more towards exploring the algorithm such that equivalence queries are not asked often.

6.1.3 RQ3: How can we measure the performance of such an algorithm and how does this algorithm perform

Performance can be measured by providing a significantly large dataset and seeing how well it performs, and how well it performs over time. In this thesis, we looked at the performance of uniformly and non-uniformly sampled datasets. We compared our algorithm with EDSM and an iMAT inspired algorithm. DAALder performed worse when the dataset does not contain enough information to construct the correct state machine, as then it takes a long time to include all traces from the dataset. When the dataset becomes much larger DAALder performs much better than EDSM as EDMS runs out of memory and starts swapping. The iMAT inspired algorithm also performs very similar as DAALder. However, this algorithm has a weakness when the data is too sparse and its observation table contains too many blanks. Thus DAALder is the best-performing algorithm when the data is too large to fit completely into memory, but is still sparse enough that it is not outperformed by an iMAT inspired algorithm.

6.2 About learning

Machines that learn try to generalize from the given data. They try to compress all the data to some form that generalizes it. In a sense, this compression is a form of *forgetting* parts of the data [18]. However, in this thesis, we can see that not only should we forget what we learn to gain insight, we should also limit ourselves to a selection of what's worth it to learn. In a sense, we should not only learn how to learn but also learn what to learn.

Bibliography

[1] Andreas Hagerer et al. “Efficient Regression Testing of CTI-Systems”. In: *Annual review of communication* 55 (2001), pp. 1033–1040.

[2] Kenneth L. Ingham et al. “Learning DFA Representations of HTTP for Protecting Web Applications”. In: *Computer Networks* 51.5 (Apr. 2007), pp. 1239–1255. ISSN: 13891286. DOI: 10.1016/j.comnet.2006.09.016. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128606002416 (visited on 11/20/2023).

[3] Clinton Cao et al. “Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster”. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. Comment: 9 pages, 12 figures, workshop paper. Aug. 2022, pp. 1–9. DOI: 10.1145/3538969.3543810. arXiv: 2207.12087 [cs]. URL: http://arxiv.org/abs/2207.12087 (visited on 09/05/2023).

[4] Chia Yuan Cho et al. “{MACE}: {Model-inference-Assisted} Concolic Exploration for Protocol and Vulnerability Discovery”. In: *20th USENIX Security Symposium (USENIX Security 11)*. 2011. URL: https://www.usenix.org/conference/usenix-security-11/mace-model-inference-assisted-concolic-exploration-protocol-and (visited on 11/20/2023).

[5] Paolo Milani Comparetti et al. “Prospex: Protocol Specification Extraction”. In: *2009 30th IEEE Symposium on Security and Privacy*. May 2009, pp. 110–125. DOI: 10.1109/SP.2009.14. URL: https://ieeexplore.ieee.org/abstract/document/5207640 (visited on 11/20/2023).

[6] Joao Antunes, Nuno Neves, and Paulo Verissimo. “Reverse Engineering of Protocols from Network Traces”. In: *2011 18th Working Conference on Reverse Engineering*. Limerick, Ireland: IEEE, Oct. 2011, pp. 169–178. ISBN: 978-1-4577-1948-6. DOI: 10.1109/WCRE.2011.28. URL: http://ieeexplore.ieee.org/document/6079839/ (visited on 11/20/2023).

[7] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. “Discoverer: Automatic Protocol Reverse Engineering”. In: *USENIX Security Symposium* (2007), pp. 1–14.

[8] Antonia Bertolino et al. “Automatic Synthesis of Behavior Protocols for Composable Web-Services”. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE ’09. New York, NY, USA: Association for Computing Machinery, Aug. 2009, pp. 141–150. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595719. URL: https://dl.acm.org/doi/10.1145/1595696.1595719 (visited on 11/20/2023).

[9] Glenn Ammons, Rastislav Bodík, and James R. Larus. “Mining Specifications”. In: *ACM SIGPLAN Notices* 37.1 (Jan. 2002), pp. 4–16. ISSN: 0362-1340. DOI: 10.1145/565816.503275. URL: https://dl.acm.org/doi/10.1145/565816.503275 (visited on 11/20/2023).

[10] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. “Combining Model Learning and Model Checking to Analyze TCP Implementations”. In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 454–471. ISBN: 978-3-319-41540-6. DOI: 10.1007/978-3-319-41540-6_25.

[11] Paul Fiterău-Broştean et al. “Model Learning and Model Checking of SSH Implementations”. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. SPIN 2017. New York, NY, USA: Association for Computing Machinery, July 2017, pp. 142–151. ISBN: 978-1-4503-5077-8. DOI: 10.1145/3092282.3092289. URL: https://dl.acm.org/doi/10.1145/3092282.3092289 (visited on 09/15/2023).

[12] E Mark Gold. “Complexity of Automaton Identification from given Data”. In: *Information and Control* 37.3 (June 1978), pp. 302–320. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(78)90562-4. URL: https://www.sciencedirect.com/science/article/pii/S0019995878905624 (visited on 12/22/2023).

[13] Georgios Giantamidis, Stavros Tripakis, and Stylianos Basagiannis. “Learning Moore Machines from Input–Output Traces”. In: *International Journal on Software Tools for Technology Transfer* 23.1 (Feb. 2021), pp. 1–29. ISSN: 1433-2787. DOI: 10.1007/s10009-019-00544-0. URL: https://doi.org/10.1007/s10009-019-00544-0 (visited on 12/21/2023).

[14] Mark Moeller et al. “Automata Learning with an Incomplete Teacher”. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 21:1–21:30. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.21. URL: https://drops.dagstuhl.de/opus/volltexte/2023/18214 (visited on 10/12/2023).

[15] John Horton Conway. *Regular Algebra and Finite Machines*. Courier Corporation, 1966. ISBN: 978-0-486-31058-9.

[16] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. “Introduction to Automata Theory, Languages, and Computation, 2nd Edition”. In: *ACM SIGACT News* 32.1 (Mar. 2001), pp. 60–65. ISSN: 0163-5700. DOI: 10.1145/568438.568455. URL: https://dl.acm.org/doi/10.1145/568438.568455 (visited on 09/05/2023).

[17] Dana Angluin. “Learning Regular Sets from Queries and Counterexamples”. In: *Information and Computation* 75.2 (Nov. 1987), pp. 87–106. ISSN: 08905401. DOI: 10.1016/0890-5401(87)90052-6. URL: https://linkinghub.elsevier.com/retrieve/pii/0890540187900526 (visited on 09/15/2023).

[18] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. CUP, 2010.

[19] Muzammil Shahbaz and Roland Groz. “Inferring Mealy Machines”. In: *FM 2009: Formal Methods*. Ed. by Ana Cavalcanti and Dennis R. Dams. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 207–222. ISBN: 978-3-642-05089-3. DOI: 10.1007/978-3-642-05089-3_14.

[20] Ronald L Rivest and Robert E Schapire. “Inference of Finite Automata Using Homing Sequences”. In: *Proceedings of the twenty-first annual ACM symposium on Theory of computing* (1989), pp. 411–420.

[21] François Coste and Daniel Fredouille. “Unambiguous Automata Inference by Means of State-Merging Methods”. In: *Machine Learning: ECML 2003*. Ed. by Nada Lavrač et al. Berlin, Heidelberg: Springer, 2003, pp. 60–71. ISBN: 978-3-540-39857-8. DOI: 10.1007/978-3-540-39857-8_8.

[22] Jos’e Oncina and Pedro Garcia. “Identifying Regular Languages in Polynomial Time”. In: *Advances in Structural and Syntactic Pattern Recognition* (1992), pp. 99–108. URL: https://www.worldscientific.com/doi/abs/10.1142/9789812797919_0007 (visited on 11/17/2023).

[23] 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. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 1–12. ISBN: 978-3-540-68707-8. DOI: 10.1007/BFb0054059.

[24] Sicco Verwer and Christian A. Hammerschmidt. “Flexfringe: A Passive Automaton Learning Package”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2017, pp. 638–642. DOI: 10.1109/ICSME.2017.58. URL: https://ieeexplore.ieee.org/abstract/document/8094471 (visited on 11/22/2023).

[25] Sicco Verwer and Christian Hammerschmidt. *FlexFringe: Modeling Software Behavior by Learning Probabilistic Automata*. 2022. DOI: 10.48550/arXiv.2203.16331. arXiv: 2203.16331 [cs]. URL: http://arxiv.org/abs/2203.16331 (visited on 09/06/2023).

[26] Neil Walkinshaw et al. “STAMINA: A Competition to Encourage the Development and Assessment of Software Model Inference Techniques”. In: *Empirical Software Engineering* 18.4 (Aug. 2013), pp. 791–824. ISSN: 1573-7616. DOI: 10.1007/s10664-012-9210-3. URL: https://doi.org/10.1007/s10664-012-9210-3 (visited on 12/13/2023).

[27] M.Y. Eltabakh, R. Eltarras, and W.G. Aref. “Space-Partitioning Trees in PostgreSQL: Realization and Performance”. In: *22nd International Conference on Data Engineering (ICDE’06)*. Apr. 2006, pp. 100–100. DOI: 10.1109/ICDE.2006.146.

[28] Gernot A. Fink. "N-Gram Models". In: *Markov Models for Pattern Recognition: From Theory to Applications*. Ed. by Gernot A. Fink. Advances in Computer Vision and Pattern Recognition. London: Springer, 2014, pp. 107–127. ISBN: 978-1-4471-6308-4. DOI: 10.1007/978-1-4471-6308-4_6. URL: https://doi.org/10.1007/978-1-4471-6308-4_6 (visited on 11/16/2023).

[29] Junghoo Cho and S. Rajagopalan. "A Fast Regular Expression Indexing Engine". In: *Proceedings 18th International Conference on Data Engineering*. Feb. 2002, pp. 419–430. DOI: 10.1109/ICDE.2002.994755. URL: https://ieeexplore.ieee.org/abstract/document/994755?casa_token=NrD6BYutxvcAAAAA:Cp1JfMOgMsXuVLe1hcymTJPPQWdd7VnkK7CZoxqwATIOvFeJibTlSoMQkzvp0fQ_OTKJ_BlhYQ (visited on 05/06/2024).

[30] Peter Eisentraut et al. *PostgreSQL 15.6 Documentation*. Nov. 2023. URL: https://www.postgresql.org/docs/15/index.html (visited on 12/01/2023).

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. Apr. 2013, pp. 38–49. DOI: 10.1109/ICDE.2013.6544812. URL: https://ieeexplore.ieee.org/abstract/document/6544812 (visited on 12/01/2023).

[32] Rafael C. Carrasco and Jose Oncina. "Learning Stochastic Regular Grammars by Means of a State Merging Method". In: *Grammatical Inference and Applications*. Ed. by Rafael C. Carrasco and Jose Oncina. Berlin, Heidelberg: Springer, 1994, pp. 139–152. ISBN: 978-3-540-48985-6. DOI: 10.1007/3-540-58473-0_144.

[33] S. E. Verwer. "Efficient Identification of Timed Automata: Theory and Practice". In: (2010). URL: https://repository.tudelft.nl/islandora/object/uuid%3A61d9f199-7b01-45be-a6ed-04498113a212 (visited on 05/14/2024).

[34] Franck Thollard, Pierre Dupont, and Colin de la Higuera. "Probabilistic DFA Inference Using Kullback-Leibler Divergence and Minimality". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. ICML '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., June 2000, pp. 975–982. ISBN: 978-1-55860-707-1. (Visited on 05/14/2024).

[35] A. W. Biermann and J. A. Feldman. "On the Synthesis of Finite-State Machines from Samples of Their Behavior". In: *IEEE Transactions on Computers* C-21.6 (June 1972), pp. 592–597. ISSN: 1557-9956. DOI: 10.1109/TC.1972.5009015. URL: https://ieeexplore.ieee.org/abstract/document/5009015 (visited on 11/22/2023).

[36] Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz. "Angluin Style Finite State Machine Inference with Non-Optimal Counterexamples". In: *Proceedings of the First International Workshop on Model Inference In Testing*. MIIT '10. New York, NY, USA: Association for Computing Machinery, July 2010, pp. 11–19. ISBN: 978-1-4503-0147-3. DOI: 10.1145/1868044.1868046. URL: https://dl.acm.org/doi/10.1145/1868044.1868046 (visited on 11/16/2023).

[37] Muhammad Naeem Irfan. "State Machine Inference in Testing Context with Long Counterexamples". In: *Verification and Validation 2010 Third International Conference on Software Testing*. Apr. 2010, pp. 508–511. DOI: 10.1109/ICST.2010.68. URL: https://ieeexplore.ieee.org/abstract/document/5477048?casa_token=pXuDDvFxuukAAAAA:

cHkH9H1U1CXl7HBcBPIQVgeY2rtqAkbQuKww2iZ-IXZoYh35vDqeo-PydmzLb2TQnJ7mgPGj (visited on 11/16/2023).

[38] Andreas Birkendorf, Andreas Böker, and Hans Ulrich Simon. "Learning Deterministic Finite Automata from Smallest Counterexamples". In: *SIAM Journal on Discrete Mathematics* 13.4 (Jan. 2000), pp. 465–491. ISSN: 0895-4801. DOI: 10.1137/S0895480198340943. URL: https://epubs.siam.org/doi/abs/10.1137/S0895480198340943 (visited on 11/16/2023).

[39] Dana Angluin. "Queries and Concept Learning". In: *Machine Learning* 2.4 (Apr. 1988), pp. 319–342. ISSN: 1573-0565. DOI: 10.1023/A:1022821128753. URL: https://doi.org/10.1023/A:1022821128753 (visited on 09/14/2023).

[40] Lev Reyzin. *Statistical Queries and Statistical Algorithms: Foundations and Applications*. Comment: 21 pages. May 2020. arXiv: 2004.00557 [cs, stat]. URL: http://arxiv.org/abs/2004.00557 (visited on 09/14/2023).

[41] Vitaly Feldman. "Statistical Query Learning". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Boston, MA: Springer US, 2008, pp. 894–897. ISBN: 978-0-387-30162-4. DOI: 10.1007/978-0-387-30162-4_401. URL: https://doi.org/10.1007/978-0-387-30162-4_401 (visited on 09/14/2023).

[42] Michael Kearns. "Efficient Noise-Tolerant Learning from Statistical Queries". In: *Journal of the ACM* 45.6 (Nov. 1998), pp. 983–1006. ISSN: 0004-5411. DOI: 10.1145/293347.293351. URL: https://dl.acm.org/doi/10.1145/293347.293351 (visited on 09/14/2023).

[43] D. Lee and M. Yannakakis. "Testing Finite-State Machines: State Identification and Verification". In: *IEEE Transactions on Computers* 43.3 (Mar. 1994), pp. 306–320. ISSN: 1557-9956. DOI: 10.1109/12.272431. URL: https://ieeexplore.ieee.org/abstract/document/272431 (visited on 11/17/2023).

[44] Markus Theo Frohme. *Active Automata Learning with Adaptive Distinguishing Sequences*. Comment: Master thesis, 97 pages. This document is closely based on the Master thesis of Markus Theo Frohme, submitted at TU Dortmund University, Germany on September 21st, 2015 and may be used as the reference for the "ADT" algorithm. Feb. 2019. DOI: 10.48550/arXiv.1902.01139. arXiv: 1902.01139 [cs, stat]. URL: http://arxiv.org/abs/1902.01139 (visited on 11/17/2023).

[45] O. Maler and A. Pnueli. "On the Learnability of Infinitary Regular Sets". In: *Information and Computation* 118.2 (May 1995), pp. 316–326. ISSN: 0890-5401. DOI: 10.1006/inco.1995.1070. URL: https://www.sciencedirect.com/science/article/pii/S089054018571070X (visited on 11/16/2023).

[46] George Argyros and Loris D'Antoni. "The Learnability of Symbolic Automata". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 427–445. ISBN: 978-3-319-96145-3. DOI: 10.1007/978-3-319-96145-3_23.

[47] Oded Maler and Irini-Eleftheria Mens. “A Generic Algorithm for Learning Symbolic Automata from Membership Queries”. In: *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*. Ed. by Luca Aceto et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 146–169. ISBN: 978-3-319-63121-9. DOI: 10.1007/978-3-319-63121-9_8. URL: https://doi.org/10.1007/978-3-319-63121-9_8 (visited on 09/15/2023).

[48] Sofia Cassel et al. “Active Learning for Extended Finite State Machines”. In: *Formal Aspects of Computing* 28.2 (Apr. 2016), pp. 233–263. ISSN: 1433-299X. DOI: 10.1007/s00165-016-0355-5. URL: https://doi.org/10.1007/s00165-016-0355-5 (visited on 11/16/2023).

[49] Benedikt Bollig et al. “Angluin-Style Learning of NFA”. In: *IJCAI* 9 (2009), pp. 004–1009.

[50] Warawoot Pacharoen et al. “Active Learning of Nondeterministic Finite State Machines”. In: *Mathematical Problems in Engineering* 2013 (Dec. 2013), e373265. ISSN: 1024-123X. DOI: 10.1155/2013/373265. URL: https://www.hindawi.com/journals/mpe/2013/373265/ (visited on 09/20/2023).

[51] Malte Isberner, Falk Howar, and Bernhard Steffen. “The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning”. In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 307–322. ISBN: 978-3-319-11164-3. DOI: 10.1007/978-3-319-11164-3_26.

[52] Frits Vaandrager et al. “A New Approach for Active Automata Learning Based on Apartness”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 223–243. ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_12.

[53] Frits Vaandrager et al. *A New Approach for Active Automata Learning Based on Apartness*. Jan. 2022. DOI: 10.48550/arXiv.2107.05419. arXiv: 2107.05419 [cs]. URL: http://arxiv.org/abs/2107.05419 (visited on 09/15/2023).

[54] Rick Smetsers, Joshua Moerman, and David N. Jansen. “Minimal Separating Sequences for All Pairs of States”. In: *Language and Automata Theory and Applications*. Ed. by Adrian-Horia Dediu et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 181–193. ISBN: 978-3-319-30000-9. DOI: 10.1007/978-3-319-30000-9_14.

[55] José L. Balcázar et al. “Algorithms for Learning Finite Automata from Queries: A Unified View”. In: *Advances in Algorithms, Languages, and Complexity*. Ed. by Ding-Zhu Du and Ker-I Ko. Boston, MA: Springer US, 1997, pp. 53–72. ISBN: 978-1-4613-3394-4. DOI: 10.1007/978-1-4613-3394-4_2. URL: https://doi.org/10.1007/978-1-4613-3394-4_2 (visited on 09/14/2023).

[56] Olga Grinchtein and Martin Leucker. “Learning Finite-State Machines from Inexperienced Teachers”. In: *Grammatical Inference: Algorithms and Applications*. Ed. by Yasubumi Sakakibara et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 344–345. ISBN: 978-3-540-45265-2. DOI: 10.1007/11872436_30.

[57] Martin Leucker and Daniel Neider. “Learning Minimal Deterministic Automata from Inexperienced Teachers”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 524–538. ISBN: 978-3-642-34026-0. DOI: 10.1007/978-3-642-34026-0_39.

[58] Olga Grinchtein, Martin Leucker, and Nir Piterman. “Inferring Network Invariants Automatically”. In: *Automated Reasoning*. Ed. by David Hutchison et al. Vol. 4130. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 483–497. ISBN: 978-3-540-37187-8 978-3-540-37188-5. DOI: 10.1007/11814771_40. URL: http://link.springer.com/10.1007/11814771_40 (visited on 01/08/2024).

[59] Yu-Fang Chen et al. “Learning Minimal Separating DFA’s for Compositional Verification”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Stefan Kowalewski and Anna Philippou. Vol. 5505. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 31–45. ISBN: 978-3-642-00767-5 978-3-642-00768-2. DOI: 10.1007/978-3-642-00768-2_3. URL: http://link.springer.com/10.1007/978-3-642-00768-2_3 (visited on 01/08/2024).

[60] J. A. Brzozowski and E. J. McCluskey. “Signal Flow Graph Techniques for Sequential Circuit State Diagrams”. In: *IEEE Transactions on Electronic Computers* EC-12.2 (Apr. 1963), pp. 67–76. ISSN: 0367-7508. DOI: 10.1109/PGEC.1963.263416. URL: http://ieeexplore.ieee.org/document/4037802/ (visited on 03/13/2024).

[61] Dean N. Arden. “Delayed-Logic and Finite-State Machines”. In: *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961)*. Oct. 1961, pp. 133–151. DOI: 10.1109/FOCS.1961.13. URL: https://ieeexplore.ieee.org/abstract/document/5397289 (visited on 03/13/2024).

[62] S. C. Kleene. “Representation of Events in Nerve Nets and Finite Automata”. In: *Representation of Events in Nerve Nets and Finite Automata*. Princeton University Press, 1956, pp. 3–42. ISBN: 978-1-4008-8261-8. DOI: 10.1515/9781400882618-002. URL: https://www.degruyter.com/document/doi/10.1515/9781400882618-002/pdf?licenseType=restricted (visited on 03/13/2024).

[63] R. McNaughton and H. Yamada. “Regular Expressions and State Graphs for Automata”. In: *IRE Transactions on Electronic Computers* EC-9.1 (Mar. 1960), pp. 39–47. ISSN: 0367-9950. DOI: 10.1109/TEC.1960.5221603. URL: https://ieeexplore.ieee.org/document/5221603 (visited on 03/13/2024).

[64] Yo-Sub Han. “State Elimination Heuristics for Short Regular Expressions”. In: *Fundamenta Informaticae* 128.4 (2013), pp. 445–462. ISSN: 01692968. DOI: 10.3233/FI-2013-952. URL: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/FI-2013-952 (visited on 03/13/2024).

[65] Simon Dieck and Sicco Verwer. “Learning Syntactic Monoids from Samples by Extending Known Algorithms for Learning State Machines”. In: *Proceedings of 16th Edition of the International Conference on Grammatical Inference*. PMLR, July 2023, pp. 59–79. URL: https://proceedings.mlr.press/v217/dieck23a.html (visited on 03/11/2024).

[66] Hermann Gruber and Markus Holzer. “Finite Automata, Digraph Connectivity, and Regular Expression Size”. In: *Automata, Languages and Programming*. Ed. by Luca Aceto et al. Vol. 5126. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 39–50. ISBN: 978-3-540-70582-6 978-3-540-70583-3. DOI: 10.1007/978-3-540-70583-3_4. URL: http://link.springer.com/10.1007/978-3-540-70583-3_4 (visited on 03/13/2024).

[67] Dana Angluin. “Inductive Inference of Formal Languages from Positive Data”. In: *Information and Control* 45.2 (May 1980), pp. 117–135. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(80)90285-5. URL: https://www.sciencedirect.com/science/article/pii/S0019995880902855 (visited on 04/10/2024).

[68] Liu Liu et al. “Insider Threat Identification Using the Simultaneous Neural Learning of Multi-Source Logs”. In: *IEEE Access* 7 (2019), pp. 183162–183176. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2957055. URL: https://ieeexplore.ieee.org/abstract/document/8918248 (visited on 03/11/2024).

[69] Rihan Hai et al. “Amalur: Data Integration Meets Machine Learning”. In: *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. Apr. 2023, pp. 3729–3739. DOI: 10.1109/ICDE55515.2023.00301. URL: https://ieeexplore.ieee.org/document/10184649 (visited on 11/22/2023).