Universiteit Leiden
The Netherlands

Bachelor Data Science & Artificial Intelligence

Benchmarking

Automata Learning

Ivan Bichev

First supervisor:
Marcello Bonsangue
Second supervisor:
Tobias Kappé

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 1, 2025

Abstract

Test data that thoroughly exercises different learners is needed to evaluate automata learning algorithms. This thesis implements and assesses several string generation techniques (random walks with adaptive termination probability, uniform sampling, and a simple-path heuristic) to create such test suites. We conduct multiple experiments based on minimized DFAs, which we use to benchmark passive automata learning algorithms (RPNI, EDSM) and an active one (a restricted version of the L^* active learner). Results demonstrate that the most varied datasets are those generated by the simple-path coverage. In most cases, we can see that EDSM performs better than RPNI, while our version of L^* with a data-oriented equivalence oracle still attains 96% accuracy. Runtime comparisons reveal that L^* scales best with DFA size, followed by a modified version of RPNI. The results confirm the integrity of the benchmark suite and provide recommendations for subsequent testing of neural and symbolic automaton learners.

Contents

Chapter 1

Introduction

Benchmarking is the systematic evaluation of a system's performance, whether physical or not, under specific controlled and comparable conditions. In principle, it requires defining a common baseline and then measuring both efficiency (for example, execution time or resource usage) and accuracy (i.e, error rates or correctness) across multiple systems or processes. In algorithmic learning, the baseline typically takes the form of an input dataset, which consists of a pre-generated collection of examples (often called a "benchmark" dataset). That way, we can ensure that observed outcome differences arise solely from the learners' internal mechanisms.

This thesis focuses on finite-state-machine inference, where various algorithms can reconstruct a target system's behaviors modeled as finite, directed, and labeled graphs (finite automata). These methods range from classical symbolic approaches, such as state-merging, to the more recent neural-network-based techniques. A robust benchmark dataset is essential in this context, since it must thoroughly exercise each learner, revealing how effectively it recovers the automaton's states, transitions, and the language accepted under different scenarios.

While other benchmark suites for automata learning exist [1, 2], they often rely on simple random sampling to generate input strings and don't offer much in terms of differing ratios of accepted/rejected data. While randomness can uncover common behaviors, it frequently misses corner-case structures and rarely stresses the full complexity of the system under learning. In Chapter 3, we examine these limitations and argue that more deliberate, heuristic-driven generation strategies might perform better. In Chapter 4, we discuss the implementation of the so-called simple path approach, which significantly improves the capabilities of learning algorithms, compared to current practices.

Beyond dataset construction, this study also benchmarks a range of inference algorithms on the newly generated inputs, measuring their performance.

This thesis explores the following research questions: The central question is:

> **RQ_0: How to generate a benchmark for testing the quality of different active and passive automaton learning algorithms?**

To answer RQ_0, it has to be decomposed into three more specific sub-questions:

RQ_1: What algorithms for string generation provide the most diverse datasets? A high-quality benchmark requires input sequences that exercise learners over various behaviors. Here, we survey existing string-generation techniques (e.g.,

fixed-termination probability random walks) and novel ones (for example, simple path traversal) to measure their ability to produce data with maximal state and transition coverage.

RQ_2: How do different automata learning algorithms compare in terms of accuracy and training efficiency?
With the diverse dataset in hand, both active (e.g., Angluin's L^*[3]) and passive (e.g., RPNI[4], EDSM[1]) learners reconstruct different target machines. We assess their performance along two axes:

- *Accuracy*: how closely the learned model matches the ground truth (e.g., via state-equivalence or error-rate metrics).
- *Efficiency*: computational cost in terms of queries, membership/test counts, but mostly in runtime.

RQ_3: To what degree do the benchmarking results from previous research align with our own?
Finally, after performing the experiments, we compare the outcomes to those from current literature and other benchmark platforms. This comparison has the goals of: (a) validating the newly created suite against established studies and (b) highlighting any discrepancies or novel insights that emerge when using the new input sets.

1.1 Applications of Automata Learning

Various domains have employed automata learning because finite-state models can concisely capture sequential or reactive behavior. Thus, one can portray almost any well-studied system (to a certain extent) and reduce it to an automaton, making it predictable and easily interpretable. Below are some of the key application areas of automata learning, ranging from a wide variety of domains.

The first and maybe most common application is software verification and model-based testing. Active learning techniques such as Angluin's L^* algorithm enable the inference of a black-box software component's behavior by posing membership and equivalence queries. The resulting automaton serves as a reference model for conformance testing. Thanks to such testing, many bugs have been reported and fixed in different binary protocols, for example, Bluetooth[5]. Another use is learning a **git** version control system [6]. Accordingly, the use of such techniques has found its way in the domain of cybersecurity and analysis of internet protocols[7].

Another application of finite-state-machine inference is in the sphere of Biological Sequence Analysis. Searls demonstrates that regular grammars can describe gene regulatory elements, laying the groundwork for automated motif discovery [8]. Sequences of nucleotides and amino acids exhibit recurring patterns that finite-state models can capture. Sakakibara reviews stochastic grammar inference methods for DNA/RNA, using probabilistic transitions to predict functional motifs and secondary structures [9].

An interesting use of automata learning is when it complements machine learning. Recurrent Neural Network models (RNNs) can perform well on sequential tasks; however, their internal decision logic is often opaque. Automata learning enables clarity by extracting finite-state abstractions from

trained networks. Discrete-time RNNs have been proven to yield human-readable rule sets [10].

A different application exists in the field of robotics, or specifically in the field of reinforcement learning. A finite automaton can represent a robot's environment, helping in guiding exploration and map construction. Furthermore, using finite-state machines, one can better visualize and interpret different learned policies[11, 12].

Finally, since automata often represent regular languages, they can be helpful in the context of Natural Language Processing. While a whole natural language requires context-sensitive formalisms, many sub-tasks such as tokenization, morphology, and simple parsing, can be addressed with regular or stochastic context-free grammars[13].

1.2 Thesis Overview

To answer our research questions, this thesis consists of five additional chapters.

Chapter 2 introduces the essential terminology and notation of automata theory and describes the different learning algorithms. It also presents additional concepts, such as probability decay functions and basic graph traversal algorithms, that form the theoretical foundation for the string database generation.

In Chapter 3, there is an overview of existing work on automata-based data generation and learning. There, we summarize the major streams in scientific literature by offering a critical evaluation of each approach and pinpointing specific gaps in current data generation methods. The chapter concludes by explaining how this thesis's contributions aim to fill in those gaps.

Chapter 4 describes the implementation framework developed for this study. It covers the construction and modification of the automata, details the novel data-generation techniques explored here, and lays out the experimental setup. Finally, it defines how the evaluation metrics will assess all the results.

Chapter 5 presents and analyzes the experimental findings. We compare our outcomes to those reported in prior studies, noting any deviations or anomalies. Then, a short section reflects on how the results relate to the research questions above.

Chapter 6 summarizes the contributions of this thesis, reflects on its limitations, and proposes directions for future work.

Chapter 2

Background Theory

To build a solid foundation for the experiments and methods that follow, this chapter introduces the theoretical prerequisites for automata inference. It defines key concepts in automata theory, outlines major learning paradigms (passive and active), and introduces a few supporting tools.

2.1 Key Terms

2.1.1 Automata Theory

In this subsection, we establish the formal definitions from automata theory that will underpin the rest of this work. Organized for clarity and precision, these establish a common notation and terminology, ranging from alphabets and strings to deterministic finite automata and their minimization, that we will rely on in later chapters. A basis for those is the book *"Automata, Computability and Complexity: Theory and Applications"* by E. Rich[14].

Alphabet (Σ) A finite, non-empty set of symbols. All strings and languages in this work are built over Σ.

The most common alphabet sizes we use in our experiments are $|\Sigma| = (2, 3)$.

String A finite sequence of symbols from Σ. We write a typical string as $w = a_1 a_2 \dots a_n$, where each $a_i \in \Sigma$. The special empty string of length zero is denoted by λ.

Language (L) A set of strings over Σ, i.e. $L \subseteq \Sigma^*$. A *regular language* is any L satisfying Kleene's theorem (below).

Kleene's Theorem The following are equivalent for a language $L \subseteq \Sigma^*$:

1. L can be described by a regular expression.
2. L can be recognized by a deterministic (or nondeterministic) finite automaton.

This theorem is the cornerstone behind the Regex-based method from Section 4.2.5.

Deterministic Finite Automaton (DFA) A DFA is a 5-tuple

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, A)$$

where

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

On input $w = a_1 a_2 \dots a_n$, the machine starts in q_0 and successively applies δ; it *accepts* w exactly if the final state $\delta^*(q_0, w) \in A$.

Complete DFA (CDFA) A DFA in which $\delta(q, a)$ is defined for *every* $q \in Q$ and $a \in \Sigma$. This guarantees no "missing" transitions.

This thesis is mostly concerned with the last automaton variant since it makes generation more straightforward. For a visual representation of such structure refer to Figure 4.1a. Furthermore, any DFA can be converted to a CDFA by sending undefined arcs to rejecting states for which all of their outgoing transitions loop back to themselves (sink states).

Empty-string Acceptance The empty string λ is accepted precisely when $q_0 \in A$.

Degree of a CDFA The number of outgoing transitions from each state. For a complete deterministic finite automaton (CDFA) $\mathcal{A} = (Q, \Sigma, \delta, q_0, A)$, the degree is $|\Sigma|$, since $\delta(q, a)$ must be defined for every state $q \in Q$ and every symbol $a \in \Sigma$. In other words, a CDFA of degree d has exactly d outgoing transitions from every state.

Graph Isomorphism (for DFAs) A bijection $f : Q_1 \to Q_2$ between two DFAs $\mathcal{A}_1 = (Q_1, \dots)$ and $\mathcal{A}_2 = (Q_2, \dots)$ that preserves:

- Start state: $f(q_{0,1}) = q_{0,2}$,
- Acceptance: $q \in A_1 \iff f(q) \in A_2$,
- Transitions: $f\big(\delta_1(q, a)\big) = \delta_2\big(f(q), a\big)$ for all $a \in \Sigma$.

Minimal Automaton Among all DFAs recognizing the same language L, the minimal automaton has the smallest number of states. It is unique up to isomorphism.

We always minimize the target DFA, since this drastically simplifies the task of tracing all possible paths. Figure 4.1b shows an example minimized automaton. To achieve this, one has to do the following procedure.

Minimization Any algorithmic procedure (e.g. Hopcroft's algorithm) that transforms a DFA into its minimal equivalent.

Quotient Automaton The result of merging a partition of equivalent states in a non-minimal DFA, yielding a smaller DFA that recognizes a superset of the given L.

The construction of quotient DFAs forms the foundation of the learning process in the majority of passive inference techniques.

Regular Expression (RegEx) A formula built inductively over Σ using:

- Union: $R + S$ (strings in R or S),
- Concatenation: $R \cdot S$ (first R, then S),
- Kleene star: R^* (zero or more repetitions of R).

Here is an example regular expression $R = (a \cdot b)^* \cdot a + b \cdot (a \cdot b)^*$.
Important note: Following Kleene's theorem, every RegEx denotes a regular language and can be converted to/from a minimal DFA.

This is all of the preliminary knowledge needed to understand the concept of finite-state-machine inference studied in this thesis.

2.1.2 Automata Learning Algorithms

We evaluate two main types of automata learning algorithms. They both learn a given system (or, more abstractly - a language), a System Under Learning (SUL).

Passive Learning

Passive learning algorithms build a model of an unknown automaton based on a fixed set of example traces without any further interaction with the SUL. Given a collection of accepted by the SUL (positive) strings, augmented with ones not belonging to the language (negative), the learner seeks a minimal DFA consistent with all examples - accepting positive traces and rejecting the counterexamples given by the negative ones.

Most of the passive learning techniques begin by breaking every accepted trace into a set of prefixes (including the empty string λ) to form the states of a so-called Prefix Tree Acceptor (PTA) that recognizes exactly every positive sample. Where they differ is in the order in which they select pairs of states as candidates for merging in the process of creating a quotient DFA. In many cases, the negative samples control the generalization, thus preventing merges of incompatible states[15]. That way, the PTA generalizes into an automaton with loops, potentially accepting the language L.

The classic passive method is the RPNI (Regular Positive and Negative Inference) algorithm, which merges states in a lexicological order [4]. The pseudocode in Algorithm 1 accurately depicts the algorithm[16].

Another algorithm we test in this thesis is Evidence Driven State Merging (EDSM), which enhances RPNI by using statistical heuristics to prioritize which merges are most likely to generalize correctly[1]. The pseudo-code is as shown on Algorithm 2[16].

The final one of these more conventional methods is Blue Fringe (BF). It, too, builds on the idea of RPNI but with a slightly different heuristic than EDSM. While EDSM chooses merge candidates more arbitrarily, BF uses a coloring heuristic that ensures that one of the merged states

Algorithm 1: RPNI (Regular Positive and Negative Inference)

Input: Positive sample S_+, negative sample S_-
Output: Deterministic Finite Automaton (DFA)

Procedure *RPNI(S_+, S_-)*
  $A \leftarrow$ BuildPrefixTreeAcceptor(S_+) ; `// Construct initial automaton from positive samples`
  $K \leftarrow \{q_0\}$; `// Initialize core states with start state`
  $Fr \leftarrow \{\delta(q_0, a) \mid a \in \Sigma\}$; `// Initialize merge candidate states`
  while $Fr \neq \emptyset$ **do**
    Choose $q \in Fr$;
    foreach $p \in K$ *(in lexicographical order)* **do**
      $A_{temp} \leftarrow$ DeterministicMerge(A, p, q); `// Temporarily merge q into p`
      if $L(A_{temp}) \cap S_- = \emptyset$ **then**
        `// Check consistency with sample`
        $A \leftarrow A_{temp}$; `// Make merge permanent`
        break; `// Exit loop after first valid merge`
    if *No merges possible* **then**
      $K \leftarrow K \cup \{q\}$; `// Add unmerged state to core`
    $Fr \leftarrow \{\delta(k, a) \mid k \in K,\ a \in \Sigma\} \setminus K$; `// Update candidates`
  return A ; `// Final consistent DFA`

Algorithm 2: EDSM (Evidence-Driven State Merging)

Input: Positive samples S_+, negative samples S_-
Output: DFA

Procedure *EDSM(S_+, S_-)*
  $A \leftarrow$ BuildAugmentedPrefixTreeAcceptor(S_+, S_-)`// Also contains negative traces`
  foreach *pair* $(p, q) \in Q \times Q$ *where* $p \neq q$ **do**
    $score(p, q) \leftarrow$ ComputeCompatibilityScore(A, p, q);
  `// Precomputes compatibility scores for all state pairs`
  repeat
    $(p^*, q^*) \leftarrow \arg\max_{(p,q)} \{score(p, q) \mid score(p, q) > 0\}$;
    if *valid* (p^*, q^*) *found* **then**
      $A \leftarrow$ MergeStates(A, p^*, q^*);
  until *no positive-score merge possible*;
  return A;

is always the root of a tree, which results in a "particularly fast and simple program"[1]. To do that, it uses a color mapping at each stage of the learning process, in which the current roots (or nodes that can only be merged to) are red, while their direct children are blue, and every other node - white. We do not evaluate this approach in the rest of the thesis.

More recently, neural network-based approaches have entered the sphere of language inference. Recurrent architectures (LSTMs or GRU) can predict the next symbol in a sequence, effectively learning an implicit state representation. Once trained, rule extraction methods are used to derive a DFA that approximates the network's behavior[17].

Overall, passive techniques are ideal when one can collect large execution logs offline and cannot probe the system dynamically. Their main limitation lies in the sensitivity towards the quality of the provided sample. Insufficient or under-representative examples can lead to overfitting or over-generalization and the inability to correct mistaken generalizations once the initial dataset is fixed.

Active Learning

Active learning methods treat the target system as a black box that they query on demand. An illustrative example is Angluin's L* algorithm, in which the learner alternates membership queries ("Is string w accepted?") with equivalence queries—asking the SUL whether the current hypothesis is identical to the target DFA. Internally, the algorithm organizes its knowledge using an observation table, a structured matrix that records the responses to membership queries for various prefixes and suffixes. This table allows the learner to detect inconsistencies, distinguish between states, and construct a hypothesis automaton. If the hypothesis is incorrect, the first string wrongly labeled by the learned model is returned as a counterexample. Whenever the learner receives a counterexample, it updates its observation table and constructs a revised hypothesis. This process is guaranteed to converge to the minimal DFA after several queries that grow only polynomially with the size of that DFA. Each loop of prompting, construction of a hypothesis, and feedback is called a learning round [3].

In practice, because the SUL behaves as a black box, researchers often replace true equivalence queries with conformance testing by generating a suite of test cases and executing them against the system. The learned model is sufficient as long as it passes all the tests [5].

Active approaches are particularly powerful when the system under test can respond quickly to queries and when high model accuracy is required. However, if queries are expensive (e.g., each test involves a time-consuming simulation) or if the system does not support equivalence testing, active learning may become impractical or simply impossible.

Some recent approaches start by building a model from existing string databases and then use a few targeted queries (prompts to the SUL) to clarify the uncertain parts. Therefore, pairing cheap passive learning with precise active one, these hybrid methods make automata inference practical even in complex settings [5].

2.2 Notation

This section presents a summary of all the notation we use throughout this thesis in the form of Table 2.1.

Symbol	Meaning
L	A given language
Σ	The set of symbols belonging to a language
a	An input symbol
Q	A set containing all the states of a DFA
q	A single state of a DFA
$\delta(q1, a)$	Transition function for a DFA
A	A set of all accepting states of a DFA
λ	The empty string
$*$	Kleene star operator
B	A subset of Q in a quotient DFA
π	A partition of a set
u	A prefix of a trace
w	A trace
n	The number of states in a DFA

Table 2.1: Notation used throughout the report.

2.3 Additional Concepts

Here are some brief definitions of terminology that appear in Chapter 4.

- **Simple Path:** A sequence of distinct vertices (or states in a DFA) $v_0, v_1, \ldots, v_k$ in a graph such that each consecutive pair (v_i, v_{i+1}) is connected by an edge, and no vertex appears more than once.
  On this definition, later on, we build our most promising heuristic.

- **Breadth-First Search (BFS):** A graph traversal technique that, starting from a source state, explores all its immediately connected by transitions states (neighbors) first, then their neighbors, and so on. It uses a queue to ensure that vertices are visited in order of increasing distance from the source.

- **Depth-First Search (DFS):** A graph traversal that starts from a source state and explores as far along each branch as possible before backtracking. Typically implemented with a stack, it visits a node's unvisited descendants before moving to its siblings. We update a set of visited states to avoid getting stuck in loops.

- **Probability Decay Functions:** Functions $d : N \to [0, 1]$ that assign a weight to each step or distance n, where $d(0) = 1$ and $d(n)$ decreases (often exponentially or polynomially) as n increases. They model diminishing influence or relevance of events the farther they lie from an origin.
  These are the basis for the newly implemented adaptive termination walk approaches introduced in the methodology part of the thesis.

- **Depth of a DFA:** The maximum over all nodes q of the shortest path length from the root to q [1].

Chapter 3

Previous Research

This chapter provides an overview of existing work related to automata learning and benchmarking. It surveys prominent literature, identifies gaps in current data generation and learner evaluation approaches, and explains how this thesis aims to address those shortcomings through novel methods.

3.1 Related Work

Active vs. Passive

This thesis draws part inspiration from the work of Aichernig et al.[5], which compares active and passive automata learning for network protocols such as Bluetooth Low Energy (BLE) and Message Queuing Telemetry Transport(MQTT). For this purpose, the authors work with Mealy machines, which, unlike DFAs, are finite-state machines where transitions have both input and output as labels.
The paper explores three research questions, which are central to its evaluation:

(RQ1) "Can passive learning based on a random sample outperform active learning?"
(RQ2) "Does the considered active automata learning algorithm generate an optimal sample?"
(RQ3) "Can random sampling support active automata learning?"[5]

Regarding RQ1, the findings indicate that active automata inference methods are superior to passive ones like RPNI, especially for the MQTT case study. While passive learning shows high accuracy (around 99%) for MQTT, the size of the learned machines is considerably larger than the ground truth models, suggesting that sparse data prevents optimal merging. For passive learning to achieve a score of 100% with random samples, it requires significantly larger data sets with longer traces than active learning. Even when the input set size is nine times larger and the trace length approximately nine times longer for MQTT, passive learning still struggles to learn correctly. We must emphasize that while active inference demonstrates outstanding performance, it may not always be practical in scenarios where querying the SUL is costly or limited. In such cases, passive learning remains a feasible option.

The other central part of the paper is the attempt to minimize the size of the data set used for passive learning. The authors achieve this by optimizing the dataset generated by the L^* algorithm. This involves identifying and removing unneeded queries and non-essential parts of the characterization set. Thus, the active learning sample size experiences a reduction by an average of

76% for BLE and 78% for MQTT. Additionally, the average trace length decreases by 67% for BLE and 43% for MQTT. With this optimized data, passive learning can correctly learn the minimal model of the SUL for all examples[5].

Lastly, the paper explores supporting active automata learning with a cache initialized by random samples. The point is to reduce the explicit calls upon the SUL by instead retrieving observations from a cache. However, this approach shows limited success: the randomly selected data captures only a minority of the data required by active learning. On average, only 27% of the data needed for correct learning in the BLE case study is present in the cache, and only 10.6% for MQTT. These results indicate that while random data can support active learning by reducing some redundant queries, it cannot replace the comprehensive exploration provided by active learning[5].

MLRegTest

This study introduces a benchmark designed to evaluate how well machine learning models, particularly neural networks, can learn regular languages - a more novel passive learning technique explained in the previous section. It tests generalization capabilities by exposing the models to a wide variety of formal languages with known logical properties (a special subset of regular languages). Using a controlled experimental setup, the study compares the performance of four neural architectures (Simple RNN, GRU, LSTM, and Transformers) across 1,800 distinct regular languages organized into 16 sub-regular classes. The goal is to assess which linguistic features and logical complexities pose the greatest challenges to current ML systems.

For each of the 1,800 languages in MLRegTest, six datasets are generated. These include training, development, and four types of test sets. Each of these has an equally balanced number of positive and negative examples, with string lengths ranging from 20 to 50 characters. The data is generated using a DFA derived from each language, while positive and negative strings are created via uniform sampling or by manipulating edit distances for adversarial examples (exclusively for the test cases to find minimal contrastive examples[5]).

The authors describe several interesting results. The uniformly random test sets achieved an overall high accuracy score (around 91%), while the more complex adversarial edit-distance-based technique resulted in accuracy scores of around 75%. This shows the model's inability to generalize near decision boundaries despite high accuracy on the random datasets used in most research. Another fact revealed by the results is that gated recurrent unit (GRU) models consistently outperform all other models across all conditions. Finally, as expected, with the increase of the training set, the test accuracy increases too[5].

There is a follow-up on this study by A. Soubki and J. Heinz[18], where they reuse the MLRegTest dataset to test different state merging algorithms (including the previously discussed RPNI and EDSM). Overall, classic passive learning techniques under-perform no matter the language category or data size. Since the original dataset lacks short strings, the authors enhance it by adding a proportional amount of short strings from 1 to 19. This modification shows a significant performance boost by increasing accuracy on the test by around 15% (for RPNI and EDSM only)[18].

Abbadingo One

The Abbadingo One is a DFA Learning Competition that took place in 1998, challenging participants to learn deterministic finite automata (DFAs) ranging from 64 to 512 states from dense and highly sparse training sets. The organizers posed sixteen problems (a 4×4 grid, size vs. sparsity). The idea was to stimulate researchers to develop new passive automata learning methods. As a result, contestants develop the EDSM and BF algorithms, which outperform the classic passive algorithm (RPNI-like)[1].

As for the dataset generation method, Lang et al. use the following procedure. To generate a DFA with an expected size of n states, a random complete DFA with $|\Sigma| = 2$ and $\frac{5}{4}n$ total states is first created. From this, they retain only the subgraph reachable from a randomly selected root state. This trimming step helps ensure that the resulting DFA has a size close to n, while also maintaining randomness in structure. Only those automata with depth exactly $2\log_2 n - 2$ are accepted to reduce variability in graph depth and simplify training set construction. This specific depth threshold corresponds to the typical maximum path length expected in such random graphs and avoids overly deep or shallow DFAs.[1]

For trace generation, a pool of binary strings is prepared with lengths ranging from 0 up to $2\log_2 n + 3$, which ensures both short and moderately long examples. As one can see, the maximum value is close to the depth of the target. The authors set the total number of strings in the pool to $16n^2 - 1$, a value empirically chosen to provide sufficient diversity and coverage of the automaton's behavior. A training set is drawn uniformly at random without replacement from this pool, while the remaining strings go in the testing set.[1]

3.2 Identified Gaps

From the current state of the art, we note the following gaps: First of all, the common ratio of accepted to rejected strings in the used datasets never changes and is usually fixed to 1:1 [2]. Thus, we suggest the exploration of systematically changing proportions of positive to negative strings. Secondly, we consider unjust the existing comparisons between active and passive methods in Aichernig et al.[5] because the active algorithm used (i.e., L^*) always converges to the equivalent SUL. A better comparison would use only randomly generated strings from the SUL when checking the equivalence, so that convergence is achieved only by a potentially infinite data set, as in the case of most passive learning methods. Finally, all the works we review in this thesis either perform random walks or take a random sample from a set of all possible strings for fixed length ranges. The only more advanced heuristic is using the L*'s queries to generate the benchmark[5, 1].

The key difference of our work is in developing better generation techniques for datasets through heuristic approaches that provide adequate coverage of a target language. Furthermore, we use random walks with novel generation techniques and differing ratios of positive to negative samples to ensure greater diversity, leading to more representative models. We also propose a novel approach involving simple path traversal, focusing on maximum coverage of transitions.

Another goal of the thesis is to assess the performance of each learning technique. The methodology section elaborates on how we achieve this by testing the different approaches.

However, this work does show some limitations. It only supports complete DFAs, and some design choices - like the length parameters for uniform sampling - may produce biased results.

In summary, by developing richer, coverage-focused benchmark datasets and systematically assessing multiple learning paradigms against them, this thesis aims to advance the state of automata-learning evaluation. The goal is to find better design guidelines for future benchmarks, for both active and passive automata learning.

Chapter 4

Methodology

In this chapter, we lay out the complete experimental framework. It explains the construction of DFAs, the generation of datasets using various string-generation techniques, and how experiments are set up to evaluate passive and active learning algorithms. Each method directly tests specific hypotheses derived from the research questions.

4.1 Dependencies Setup

This chapter begins with a detailed explanation of how the supporting structures of the current code base work.

In this thesis, we define a custom-made CDFA (Complete Deterministic Finite Automaton) object and use it as a common interface across various functionalities. This object is a Python dictionary with the following keys. First is the transition matrix, which is a 2-dimensional NumPy array. Its rows correspond to state indices, and its columns - the unique symbols of Σ. We model each directed transition in the following way: the start state is the corresponding row index, the label is the corresponding alphanumerical index of the symbol, and the cell value at that coordinate is the destination (end) state. The following elements in the dictionary are the index of the start state (set by default to 0), a list of binary values indicating whether each state is accepting or not, a corresponding list of state labels, and a list representing the elements of the input alphabet Σ.

During the creation process, the user can specify two parameters: the number of desired states and $|\Sigma|$. States are labeled sequentially from "s0" to "sn," where the number of states determines n. The alphabet is initialized with Unicode characters, beginning with the symbol "a".

The actual creation of the automaton is fairly simple. First, we initialize the symbols of Σ, the states, and their labels. Then, to construct a CDFA, for each state, there should be $|\Sigma|$ outgoing transitions, each to a random destination. To determine the terminating states, an arbitrary amount of the initialized states are selected.

In addition to the internal NumPy representation, the Python package Graphviz generates a more visual form of the automaton using the DOT language [19]. Figure 4.1a shows an example.

As mentioned, the CDFA object bridges multiple packages, and we implement several helper functions to support this integration. The current generation produces any random CDFA machine, which is rarely minimal. Working with minimal automata is preferred because:

1) It improves the efficiency of generation techniques.

2) Active learning algorithms (specifically L^*) as discussed in Chapter 2, always output minimized DFAs.

The **dfa**[20] package is responsible for the minimization procedure of the output. This package includes the **dfa.minimize()** method, which minimizes a given automaton, and **dfa.dfa2dict()**, which converts the result back into a dictionary, which is a NumPyDFA-compatible format. Figure 4.1b displays the result of applying the minimization process to the previously shown CDFA. Another useful feature of the package is **dfa.trace()**, which, given a list of input symbols, follows the trace and returns the accepting or rejecting status of the final state.

The other Python library that we use and which interacts with our implementation of a CDFA is **"AAlpy"**[21]. Other works[5] employ it since it provides tools for both active and passive learning methods. We describe those in the experiments section.

4.2 Data Generation

In this section, we explore different approaches to generating a diverse dataset of strings.

4.2.1 Random graph walks

Random walks are sufficient for the task of creating a representative enough dataset[5]. However, current implementations often terminate sequences at a random length, determined within a user-defined minimum and maximum range. While this approach is simple, it lacks control over the distribution of trace lengths. To address this issue, we implement several random walk variations that use different termination probability decay functions to ensure greater variety in trace lengths. In particular, four different termination methods described below arise, each returning two disjoint sets - one containing the accepted traces and the other containing the rejected.

Fixed termination probability

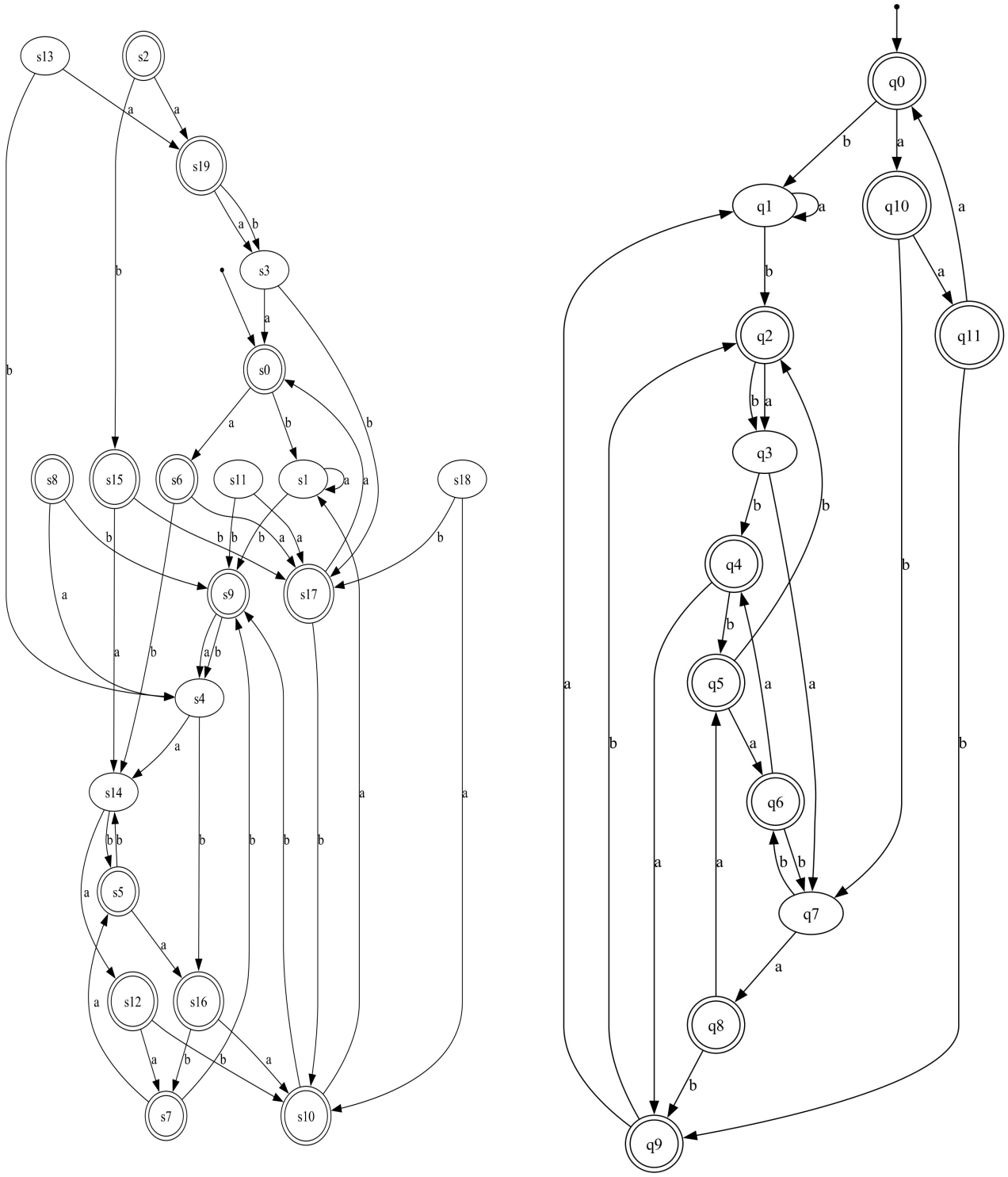
The following function serves as the baseline to evaluate the other ones' accuracy. It takes the number of runs and an optional maximum string length as arguments. This and all of the other functions share, the **account_empty** argument which if set to **True** automatically labels the λ trace. An alternative to this function is also created, which balances the accepted/rejected strings in the output to a ratio the user chooses.

Linearly decaying termination probability

This function uses linear decay to gradually decrease the termination probability after each walk. The probability for termination at each generated string is $pr_t = 1 - 0.01t$. The parameter 0.01 is a default value, which can be modified in the function call. This generator also has an alternative that supports differing proportions of accepted to rejected strings.

Root (sublinear) growing termination probability

The method emphasizes shorter to medium traces by accelerating the termination probability in the early stages of a walk. A root-growing function decreases the likelihood of longer sequences



(a) The original CDFA

(b) A minimized version of the CDFA on the left

Figure 4.1: CDFA originally generated with 20 states and alphabet length of 2. These are visual representations of automata we use in this thesis.

early on. Each traversal has a unique termination probability calculated based on the trial (walk) index. The formula we use is the following:

$$pr = \sqrt{\frac{t}{t_{max} + \frac{mode}{2}}} \tag{4.1}$$

, where t stands for current trial number, t_max stands for the **max_trials** argument, while $mode$ stands for one of the three possible **modes** provided to the function. The values for **mode** are $[0, \frac{t\_max}{2}, t\_max]$. To see the effects of different settings, refer to Figure 4.2. Again, the method supports user-specified ratio functionality.

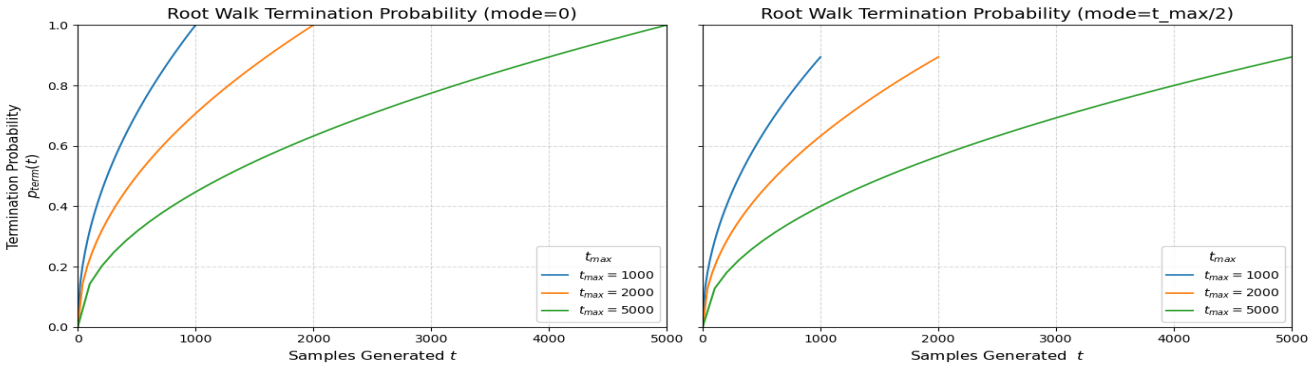


Figure 4.2: Graphs of termination probabilities for root walk in 2 modes. It is clear they favor smaller string sizes (termination probability scales fast in the beginning and then levels off).

4.2.2 Logistic growing termination probability

This method favors the traversal of longer traces over shorter ones. It achieves this using a sigmoid-like probability curve, defined by the following function:

$$p_{\text{term}}(t) = \begin{cases} 0.02, & \text{if } p_{\text{raw}}(t) \leq 0.02, \\ p_{\text{raw}}(t), & \text{if } 0.02 < p_{\text{raw}}(t) < 0.9, \\ 0.9, & \text{if } p_{\text{raw}}(t) \geq 0.9. \end{cases}$$

Where the raw probability $p_{\text{raw}}(t)$ is calculated as:

$$p_{\text{raw}}(t) = \frac{1}{1 + e^{-k\left(t - \frac{trials}{1.5}\right)}}.$$

Here, t ranges from 0 to the total number of trials, denoted by $trials$. Dividing $trials$ by 1.5 sets the sigmoid's midpoint roughly two-thirds of the way through the process. The slope parameter k (default 0.003) determines how sharply the raw probability increases, from near zero at small t, through 0.5 around $t \approx trials/1.5$, and approaching one near the end. We constrain the final probability to be between 2% and 90% to prevent extreme outputs.

Figure 4.3 plots the termination probabilities under the default parameters. As highlighted, the function's behavior depends on the size of the trial set, meaning the input parameters often need fine-tuning.

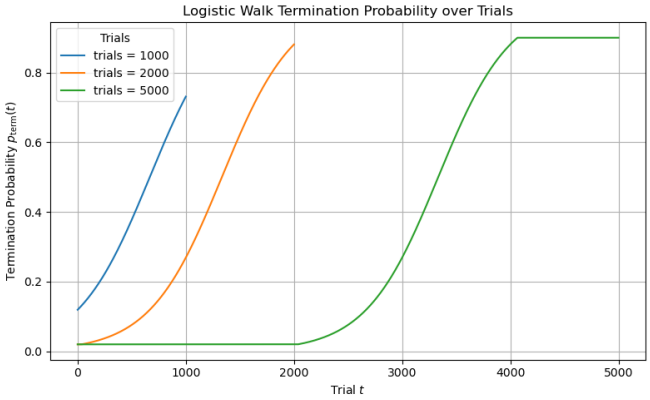


Figure 4.3: Graphs of termination probabilities for a logistic walk with default settings. This method results in a variety of longer and very short strings for larger training sets.

4.2.3 Random string selection

Next, we describe our implementation of an uniform distribution sampling.

Generating the language L

To sample random strings uniformly from any DFA, one would need to know the total size of the language, meaning the number of all possible accepted and rejected strings. However, since such a set is often infinite, generating it entirely is impossible. The practical workaround is to limit the language to a finite subset. Although we can calculate a DFA's maximum depth, doing so is computationally demanding and is not a viable option. Furthermore, tracing all possible strings through the automata takes too much time, especially with larger alphabets. To manage this, restrictions on the size of the alphabet take place when using such methods.

As for ensuring coverage of any DFA, we develop specialized implementations of both BFS and DFS, but use only the former since it produces a more natural ordering of strings. There is one important modification - the algorithm keeps track of the states and the tuple (state, sequence), ensuring uniqueness while keeping the entire history. Essentially, the idea is that with these searches, we cover all possible sequences up to the given length of the trace and alphabet. The code thus guarantees that it can generate, label, and enumerate (if needed) all target language strings up to a specified length. According to the pumping lemma, if this length equals the number of states in the minimized target automaton, it guarantees full language coverage since all non-generated strings are pumped versions of existing ones.

Due to the algorithm's big-time complexity, we use this approach sparingly, exclusively with smaller, second-degree DFAs.

Sampling the language L

For the actual sampling procedure, there are two distinct solutions. The first utilizes the above-mentioned generation technique by selecting random strings from the prepared dataset based on a uniform distribution. Given an input, this method combines the BFS search results into a

single list, which it then samples by assigning equal probability to each trace. An alternative version also allows sampling with a different fixed proportion of positive to negative samples.

The second solution draws motivation from a property of CDFAs. Since each state is always guaranteed to have $|\Sigma|$ transitions, at each step (or character of the string), each transition has the same probability of appearing, which is exactly $\frac{1}{|\Sigma|}$. Therefore, given a certain fixed length, one can generate strings following an underlying uniform distribution. There is one additional step: choosing a random string length. As the length increases, so does the probability of a representative being selected in an exponential fashion. Therefore, the "weight of a length" in this distribution has to be accounted for and can be calculated as:

$$\frac{|\Sigma|^l}{\sum_{n=0}^{l_{max}} weight(n)} \tag{4.2}$$

The only drawback of this method is that if a complete coverage set is required (as in the BFS search), each trace must be traversed through the target automaton to obtain a label, which makes the approach less efficient. Nevertheless, it is superior when the goal is to label only a few random samples.

The final approach would be to forget accurate uniform distribution and instead choose a string size, whose representatives to uniformly sample. While this might yield better results, we leave its evaluation for future work.

4.2.4 Heuristic graph traversal approaches

A good minimal dataset must fully represent the SUL. While multiple approaches exist, the most intuitive solution is to capture all transitions of the target. For this purpose, we develop an algorithm that performs a simple path search, where the goal is to find all the different simple paths up to a certain length. The pseudo-code in Algorithm 3 presents the version of the recursive approach we implement, where q stands for the state, w for a current walk (trace), and k stands for the allowed transitions left (maximum size N).

The algorithm works on the principle of DFS, going down a single branch and passing the original reference of the list to the following recursion, which fills it up. Only the list of visited states stays consistent to prevent early termination.

In principle, the maximum path length should correspond to the longest possible simple path. Therefore, the maximum value equals the number of states in the automaton. In most cases, however, the longest simple path is shorter (due to multiple dead-end states). Nevertheless, larger upper bounds do not affect the algorithm's correctness, as it does not revisit states.

4.2.5 RegEx

Another interesting approach is to use regular expressions to denote the SUL instead of an automaton. There exist algorithms for transforming DFAs into RegEx[22], and even the above mentioned **dfa** package has such a method[20].

One can work directly at the level of the Abstract Syntax Tree (AST) of the expression. Searching the AST by scanning all the branches can provide a diverse enough dataset.

However, due to time complexity and the need to simplify the resulting expressions, we have decided not to implement this method in this thesis, and we leave it for future work.

Algorithm 3: Find All Simple Paths up to Length N

Input: DFA $A = (Q, \Sigma, \delta, q_0, F)$, maximum depth N
Output: Sets *accepted*, *rejected*

```
1  Procedure PathsAllUpTo(A, N)
2      accepted, rejected ← ∅, ∅;
3      visited ← {q_0};
4      Function DFS(q, w, k)
5          if q ∈ F then
6              add (w ≠ ε) ? w : λ to accepted;
7          else
8              add (w ≠ ε) ? w : λ to rejected;
9          if k > 0 then
10             for a ∈ Σ do
11                 q' ← δ(q, a);
12                 if q' ∉ visited then
13                     add q' to visited;
14                     DFS(q', w · a, k − 1);
15                     remove q' from visited;
16     DFS(q_0, λ, N);
17     return accepted, rejected;
```

4.3 Experiment setup

This section concerns itself with an exact setup that tackles the research questions outlined in Chapter 1, as well as stating certain hypotheses related to the specific experiments.

4.3.1 Experiment 1: Evaluating the performance of different random walk methods with passive learning

The goal of the following procedure is to assess the accuracy of all the newly proposed CDFA random walk methods. This and the next two subsections directly tie in with **RQ1**.

For this task, the four random walk strategies discussed earlier are compared against one another. These are tested across 12 (4×3) categories, each learning on the same target CDFA. We set the parameters of the automata to the following: the number of states in the minimized machine is 30 and 50, with an alphabet of size 2. Additionally, we run the variation with states 50 and $|\Sigma| = 3$. The four subcategories correspond to different sample sizes (before removing duplicates). Specifically, these sizes are 100, 500, 1000, and 2500.

We choose state sizes of 30 and 50 because these values balance expressive power and computational feasibility. Automata with fewer states (e.g., under 20) tend to produce overly simple behaviors, which often fail to stress the capabilities of the learning algorithms. Conversely, significantly larger DFAs (e.g., above 100 states) can lead to serious memory and computational requirements, which make algorithms like the classic RPNI unusable. However, in some instances, CDFAs with $|Q| = 100$ are used to show the disparity of computation times between the different learners or in situations where large targets do not hinder learning speeds (e.g., active learning).

As for the alphabet size, we keep $|\Sigma|$ at 2 for most experiments to simplify the analysis and reduce the combinatorial explosion during path traversal. However, variants with $|\Sigma| = 3$ are included in some cases to quantify the effects of increased symbolic diversity. With more symbols, the same number of states leads to a wider branching factor in the state graph, which can affect the behavior of both passive and active learners by increasing transitions seen during training.
After generating the trails, we initialize the two passive learners. These are the AALpy[21] implementations of the RPNI and EDSM algorithms. RPNI has two distinct models:

1) A *classic* version, which follows the original implementation.
2) A *gsm* version (general-state-merging), which produces nearly identical results but executes faster.

Therefore, since the experiment's focus is purely on the accuracy of the algorithms, the latter variant is the optimal pick. As for EDSM, we run it with default arguments, following a similar algorithm structure as the one described in Chapter 2.

To benchmark the algorithms, at each iteration, the fixed-termination walk (with a ratio of positive to negative strings set to 1) runs for a total of 100 000 trials with the ratio of positive to negative strings set to 1 (before the removal of duplicates).

In order to obtain statistically sound results, the generating-learning cycle goes through 100 repetitions for each sample size.

4.3.2 Experiment 2: Exploring the simple path approach

This experiment setup has two parts: testing the performance of the simple path generator and making a direct comparison to random generators.

Since searching through large automata is computationally expensive, we limit the experiment to simpler CDFAs. The first part evaluates the ability of passive learning algorithms to recover key information using only simple paths. The same minimized automaton parameters from Experiment 1(4.3.1) reappear to ensure consistency with other datasets.

As for the second experiment, the focus shifts to directly comparing passive EDSM performance between the fixed-random and the new simple path datasets. The target is a minimized CDFA of size 20 with $|\Sigma| = 2$ (to reduce learning speeds). After generating the test dataset, an initial training set of five strings from each generation method acts as input to the learner. The evaluation function returns the accuracy, after which we randomly select five additional strings from the corresponding technique and add them to the training data.

4.3.3 Experiment 3: Evaluating the effect of different proportions of accepted to rejected strings on passive learning

Again, the experiment has two goals. The first objective is to evaluate the accuracy of the uniform sampling technique with different ratios compared to random walks under the same conditions. The second goal is to measure how similar modifications affect the accuracy of other traversal techniques.

Specifically, this experiment aims to determine whether it is better to use mainly positive strings, which could allow generalization, or mostly negative strings, which may help enforce stricter structural constraints. Hypothetically, including more (but not only) negative strings would preserve key elements of the Prefix Tree Acceptor (PTA), and despite the potential for slight overfitting, it could yield better overall accuracy.

The experimental setup for this part includes five versions of each sampling method: uniform selection, root termination walk, and fixed termination walk. We test each of those with positive-to-negative balances of 0, 0.25, 0.5, 0.75, and 1. For simplicity, only a single target automaton size of 50 states and $|\Sigma| = 2$ is used. For the true uniform sampling method, we consider all possible strings up to a length of 50. The test executes 100 times, with a training dataset size of 1000.

4.3.4 Experiment 4: Testing an active learning method with approximation instead of equivalence

As previously discussed, L^* performs better than passive techniques. This superiority is due to two key factors: its ability to prompt the system with specific traces and to test its hypotheses. Logically, there are two main ways to handicap the active approach.

The first would be to limit the ability to prompt the system under learning. This would effectively remove the core function of the algorithm (assuming an equivalence oracle is already available). However, if the SUL's responses are pre-generated (and thus likely do not offer full coverage), undefined labels for prompted traces become a concern. There are two possible ways the system could behave in such a case:

1) If the L^*'s prompt is undefined, automatically set it to rejected or accepted. In the first case, this could eventually lead to over-specification, or in the second, to over-generalization. Therefore, in both cases, that would lead to serious accuracy losses.
2) If the L^*'s prompt is undefined, return an unknown label, essentially forcing the algorithm to keep asking until it gets a definite response. The issue here is that currently, that would break L^* since it always expects proper feedback. Therefore, we need a new algorithm or to perform a significant modification, which is beyond the scope of this thesis.

Because of these complications, this method of limiting the algorithm is dismissed.

Alternatively, one can artificially restrict the hypothesis-checking process. This maintains the black-box nature of the SUL and aligns more closely with how conformance testing typically functions. We compare the learned language only against a finite set of traces. This approach has been explored before (e.g., in B.K. Aichernig et al. [5]), where the equivalence class resembles previously learned models. This experiment proposes a new setup: using randomly generated data (the same data as for passive techniques) as the equivalence set. Of course, L^* still has an advantage over EDSM and RPNI, but its performance is expected to degrade significantly under this constraint.

The objective of this experiment is to evaluate the impact of different equivalence oracle data sizes on the performance of the modified L^*.

As for the actual implementation in AAlpy, we first define the **"SUL"** class, which serves as an interface between the learner and the machine. The **pre()** method resets the "pointer(tracer)" to the starting state index of the Numpy CDFA. **Step()** keeps track of the current state after each transition. Since the default equivalence oracles in AALpy are designed for standard active learning and do not meet the needs of this study, we implement a custom oracle - **DataSetEqOracle()** by modifying the two default methods. In addition to its constructor - **__init__** - there is the key method **find_cex()**, which generates counterexamples by inspecting the list of accepted and rejected strings. As it tracks all queries, **find_cex()** returns the first mislabeled string it encounters. If no such string exists, it returns **None**.

Based on previous experiments, we employ the fixed-random generation technique to create the equivalence dataset. This experiment includes several categories. There are four types of L^* based on different oracle sizes: 10, 100, 1000, and 2500, for each of the three target automata from Experiment 1(4.3.1), along with an additional case (100 states, $|\Sigma| = 3$). Again, each category runs 100 times on different random seeds to ensure statistically significant results.

4.3.5 Experiment 5: Comparing training speeds of active and passive automata

Efficiency of algorithms can be a defining factor that compromises even the most accurate models. Due to the curse of dimensionality, such problems can quickly escalate from minor inconveniences to practically unsolvable scenarios. Thus, understanding which algorithms offer the fastest execution times is crucial. This experiment evaluates active (L^*) and passive (EDSM and RPNI) learning methods. This time, both versions of RPNI are present: the classic variant and the generalized state-merging version.

The actual setup is as follows. To compare active and passive approaches fairly, we pre-generate 50 randomly sampled datasets for three CDFA sizes (25, 50, and 100 states), each with $|\Sigma| = 3$, and run the learning loops. The Python **time** package measures execution time for both active and

passive approaches.

The main goal here is to address the training efficiency aspect of **RQ2**.

4.3.6 Experiment 6: Comparison to OpenFST

We directly address **RQ3**, by conducting the following experiment, where the focus is on comparing the accuracies achieved with two different automata benchmark generation techniques: (1) our custom Python-based random benchmark generator and (2) the established **OpenFST**[23] C++ library accessed via the **Pynini**[24] Python API.

We evaluate both approaches on two independent sets of 100 complete DFAs, each a minimal one with $|Q| = 50$ states over an alphabet of size $|\Sigma| = 2$. The training set consists of 100 sample traces per target. Both frameworks produce benchmarks with a balanced percentage of positive to negative traces to make the comparison as fair as possible.

Generating a training sample through **OpenFST** happens through the **RandGen()** function, which returns accepted sequences of an input DFA in the form of a PTA. It does so by randomly selecting (following a uniform distribution) a possible transition. If the current state is final, termination of the walk is added to the pool of transitions. To illustrate, if the function is at an accepting state with 2 outgoing transitions, the probability of terminating is around 33%. Since **OpenFST** does not allow sampling of negative traces, we make a complement of the target CDFA by flipping all the final states' labels and making all non-final ones accepting. Thus, we end up with 2 PTAs, which represent accepted and rejected sample sets, each having a size of 50 [23].

A procedure similar to those in the other experiments generates all the test sets (see 4.3.1).

Finally, since the chosen training sample size is relatively small, any marginal performance differences between the generation methods are difficult to detect. However, increasing the sample size significantly would make the benchmarking process too computationally intensive and memory-demanding for our system. Because of these reasons and considering the time limitation at the end stages of the thesis, there are constraints on the training size. As such, in hopes of expanding the obtained results, we leave more extensive comparisons on larger target automata and bigger data volumes for future work.

For now, as Section 5.1.7 demonstrates, the accuracy differences are minimal, but still statistically significant in some cases.

4.4 Evaluation Metrics

This section explains how this work will quantify and interpret the results. Depending on the experiment, we measure accuracy using random test datasets of various sizes. The test data accepted/rejected string proportion is always 1:1, as done in previous benchmark datasets[2]. If all of the strings in the test set pass, the learner achieves a score of 100%. Each misclassified trace (no matter its original label) deducts $\frac{1}{|D|}$ from the current score, where D stands for the test dataset. The measurements have a precision of up to 3 significant digits.

A more nuanced comparison of automata would involve calculating the graph edit distance. While this would not reflect the percentage of accepted strings, it would provide insight into the structural similarity between learned and reference automata. However, such analysis is highly resource-intensive and is unfeasible for this thesis.

Since the execution of the benchmarking procedure is both hardware and software dependent, here is a table of all the important specifications 4.1 of the system performing Experiment 5, where the times to produce results are measured in seconds.

Table 4.1: Specifications of the system used to run the experiments

Component	**Specification**
Processor	Intel Core i9-14900KF 3.2 GHz, 24 Cores, 32 Threads
Operating System	Windows 11 Home Version 10.0.26100 (Build 26100)
Memory (RAM)	32.0 GB DDR5

This chapter's focus is on discussing the different aspects of our complete benchmark generating framework as well as the theory and intuition behind the chosen methods. At the end it also states the experiment setups and justify the evaluation metrics. Thus, to round of this part of the thesis, we present a concise overview (Table 4.2) of all the used techniques.

Table 4.2: Comparison of Dataset Generation Methods

Method	**Purpose**	**Pros / Cons**	**Performance**
Uniform Sample	Classical random string sampling.	+ Simple baseline. Fast generation for CDFA. – Favors long traces.	Underwhelming performance in Exp. 3.
Fixed Walk	Baseline random walk method.	+ Simple and stable. –Limited diversity(hard to adjust for extremely large DFAs).	Best overall of the random walk methods; stable across all tested sizes.
Linear Walk	Linear length-biased termination.	+ Gradual change in string length. – Accuracy drops at scale.	5–7% worse than fixed walk.
Root Walk	Early increase in termination	+ Compact traces. – Not enough coverage, sensitive to tuning.	Worst in Exp. 1, good in Exp. 3.
Logistic Decay	Balanced trace lengths (focus on long and short)	+ Focus on short and long strings. – Sensitive to tuning.	Unstable; Comparable to linear walk.
Simple Paths	Maximize state/edge coverage.	+ Fast learning. – Needs complementary random samples to converge.	Best early-stage performance (Exp. 2).
Regex-Based	Obtain whole language and target edge cases	+ Human-readable. – Not implemented.	Not tested in this thesis.

Chapter 5

Results

This chapter presents and analyzes the results of the conducted experiments. It compares the performance of different dataset generation strategies and learning algorithms in terms of accuracy and computational efficiency. With these findings, we directly address the research questions from Chapter 1.

5.1 Overview of Findings

5.1.1 Experiment 1

The overall results from the first experiments are shown in Figure 5.1. The immediate key observation is that none of the newly implemented random walk methods achieves accuracies as high as the fixed random termination approach. The root walk has the poorest overall accuracy, 8% to 10% worse than the fixed 0.1 termination chance random walk. The other two methods, logistic and linear termination, also underperform, each showing 5% to 7% lower accuracy than the achieved maximum. However, it is worth noting that the accuracy gap between all the walk strategies tends to be smallest at the lowest train sizes (approximately 2% to 3% for size 100 in Figure 5.2) and increases significantly at bigger inputs (exceeding 13% for size 2500 in Figure 5.3). Therefore, the quality of the proposed adaptive termination probability functions may degrade as the total number of strings increases. Fine-tuning the parameters could help address this issue. To do so, we must first analyze the underlying distribution of sequence lengths outputted by each probability decay function.

As for the direct comparison of EDSM to RPNI, average performance indicates that EDSM generally outperforms RPNI (Figure 5.1), which is to be expected, given the more heuristic approach to state merging. The exception to that is the scores that learners achieve on the "root" method. Furthermore, RPNI proved to perform on par with EDSM for small datasets (Figure 5.2).

Finally, it is important to note that the achieved accuracy closely resembles the results reported in Soubki et al.[18]. However, differences in automata and benchmark techniques likely account for any deviations, as the experimental setups are not identical.

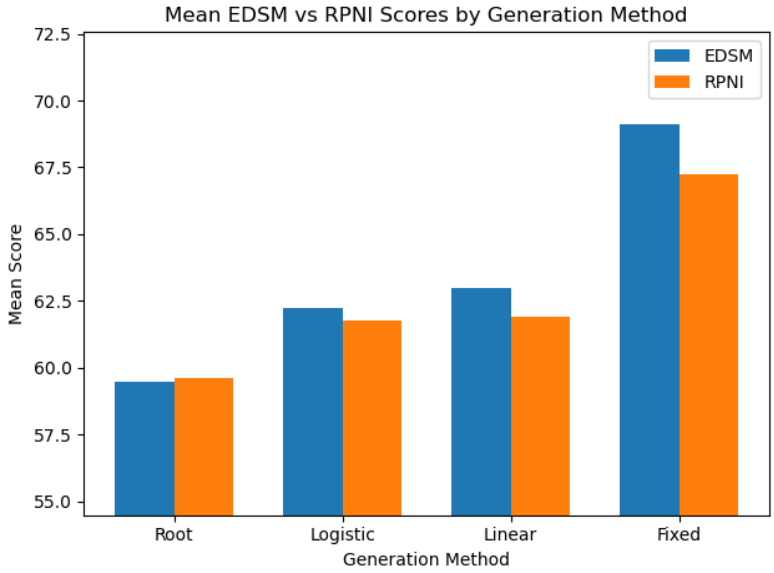


Figure 5.1: Graph of the average accuracy achieved over all experiment settings. Score is given in % and shows the advantage of EDSM over RPNI and the quality of the different sampling strategies. Fixed-termination random walk seems to generate the best string datasets.

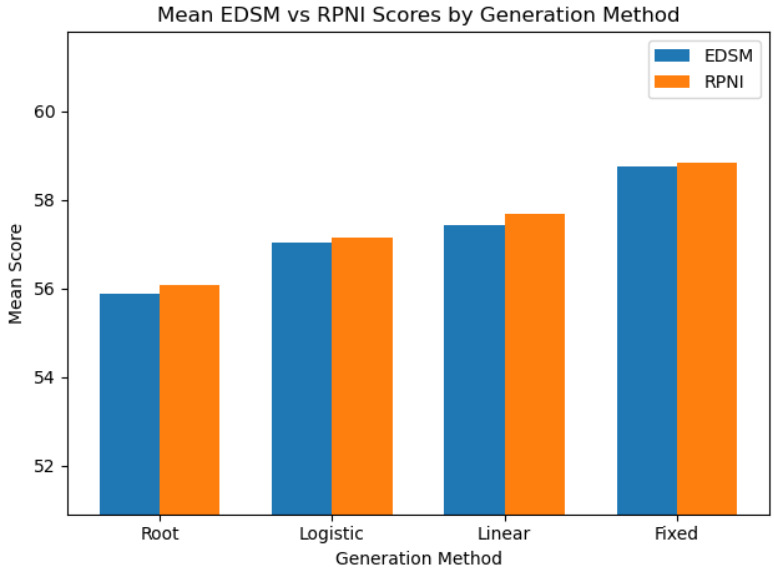


Figure 5.2: Graph of average accuracy for $|Q| = 50, |\Sigma| = 2$ for 100 samples in %. We can see that the performance of RPNI and EDSM is equalized for small datasets. RPNI has an insignificant advantage (minimum pair p-value of 0.73) - just a fraction of a %.

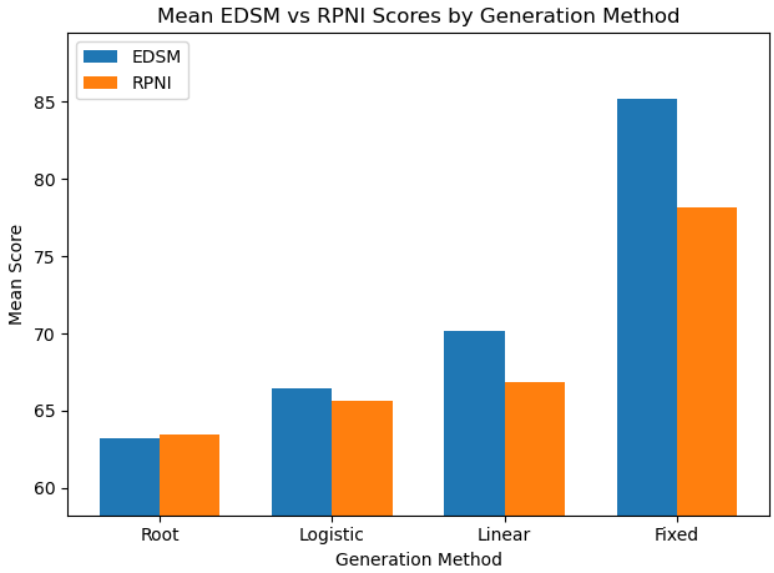


Figure 5.3: Graph of average accuracy for $|Q| = 50, |\Sigma| = 2$ for 2500 samples in %. We can see the real advantage of EDSM's score heuristic when ample and diverse data is provided.

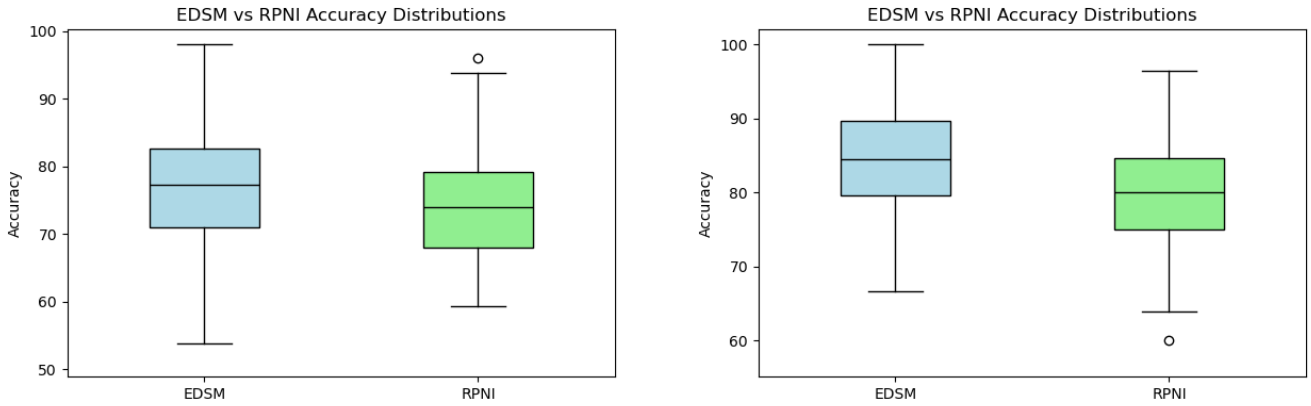
5.1.2 Experiment 2

This experiment presents the results of attempting to generate an optimal dataset using the simple path approach. From Figure 5.4, two observations are clear:

1) The new method produces high-quality results, with median accuracy reaching approximately 77% for EDSM and 75% for RPNI on smaller automata (subplot (a)), and 85% for EDSM and 82% for RPNI on larger automata (subplot (b)).
2) As expected, learners achieve higher scores on more complex CDFAs. The increase in state and transition counts makes more simple paths available for exploration. Because the dataset size scales more rapidly with larger automata, accuracy improves despite the objectively more challenging target.

A direct comparison of the efficiency of EDSM and RPNI is available on the scatter plots (Figure 5.5). The x-axis shows the sample sizes that the simple path approach produces, which is the same for both graphs. As the inserted trend lines show, overall, EDSM achieves higher accuracy.

However, the advantage of the new technique is still not truly evident. Therefore, following the description in the methodology section, we obtain the following results. The line graph in Figure 5.6 shows that after an even start, at iteration 50 (sample size of 250), the simple paths approach takes a noticeable leap in accuracy, keeping its lead until the training dataset is exhausted and the score levels at around 81%. As for the random-training-dataset learner, it takes almost twice as many strings to catch up. Since the arbitrary string approach has a potentially infinite set of strings to sample, it eventually converges to the target language. To balance that out in future experiments, one can modify the simple paths dataset by adding random strings (after thoroughly exhausting it first).



(a) $|Q| = 30, |\Sigma| = 2$ (b) $|Q| = 50, |\Sigma| = 2$

Figure 5.4: Box plots showing the distribution of 100 accuracy scores achieved with the all simple paths generation technique for two different target sizes. EDSM performs generally better, with higher accuracy for more complex automata.

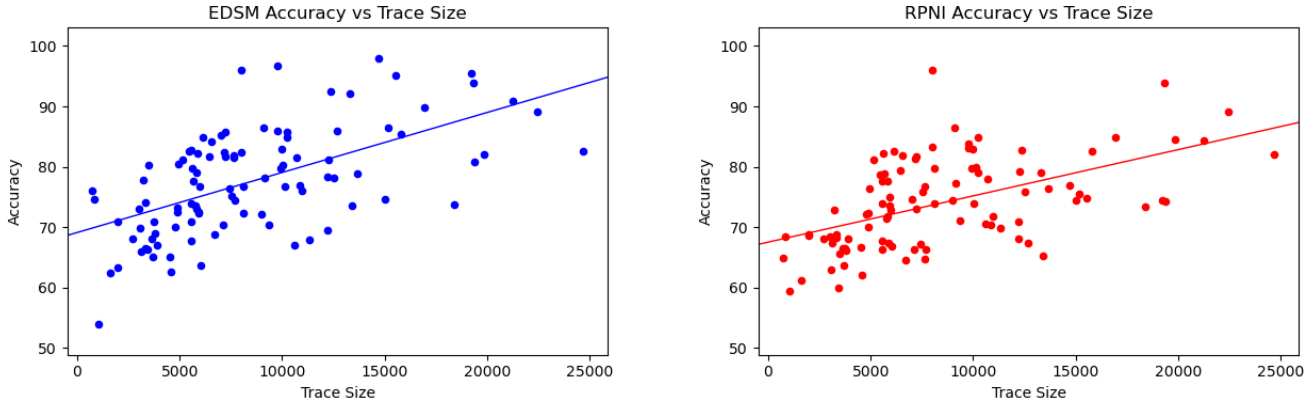


Figure 5.5: Scatter plots showing accuracy/train set size using the simple path generation heuristic, for the same targets of $|Q| = 30, |\Sigma| = 2$. The trend lines clearly show the better scaling learning capability of EDSM when combined with the new heuristic.

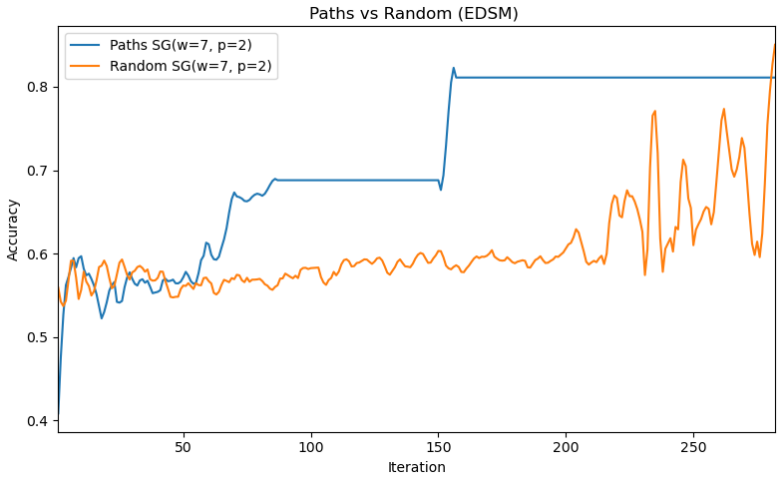


Figure 5.6: Line graphs showing the learning rates in terms of accuracy for the EDSM algorithms (with Savitzky–Golay smoothing). After acquiring a sufficient sample size, the accuracy achieved on the paths method quickly outgrows that of a fixed random walk generation. However, it eventually gets surpassed after exhausting the finite train data.

5.1.3 Experiment 3

This experiment explores how different ratios of positive to negative strings affect accuracy. From each of Figures 5.7, 5.8, and 5.9, it is evident that having a (1:1) proportion offers the best results for any of the traversal techniques we employ. It seems to provide the best generalization ability while successfully preserving constraints. Both ratios of 0 and 1 have an accuracy of 50% since the test case itself is balanced. In the first case, the PTA does not form, and there is only 1 rejecting state (extreme overfitting). In contrast, in the latter, the state merger quickly minimizes the PTA into a single accepting state (extreme underfitting).

The uniform sampling performs worse than all the other tested methods. This is likely because the traces are too large (since the maximum length is 50) and cannot provide enough information to reconstruct the target DFA [1, 18]. Thus, we suggest further investigation of this method on differing target automata sizes and maximum length parameters.

The experiment provides one more useful insight - the root walk, despite being last in Experiment 1 (5.1.1), performs better than the fixed termination walk. The accuracy is around $2-4\%$ higher. These results can be due to two reasons: either the root probability decay works exceptionally well with the 1 000 traces dataset, or the walk method lacks a good accepted-to-rejected ratio in Experiment 1.

5.1.4 Experiment 4

As expected, even though the suggested handicap prevents L^* from converging in some cases, it still performs better than EDSM by large margins despite the worst-case scenarios. The average results over all of the iterations and database sizes for the different DFA are as presented in Table 5.1. The scores for all automata are very high, with minute differences. The one case in which the algorithm struggles to achieve full target recovery is for the smallest oracle size of 10. Figure

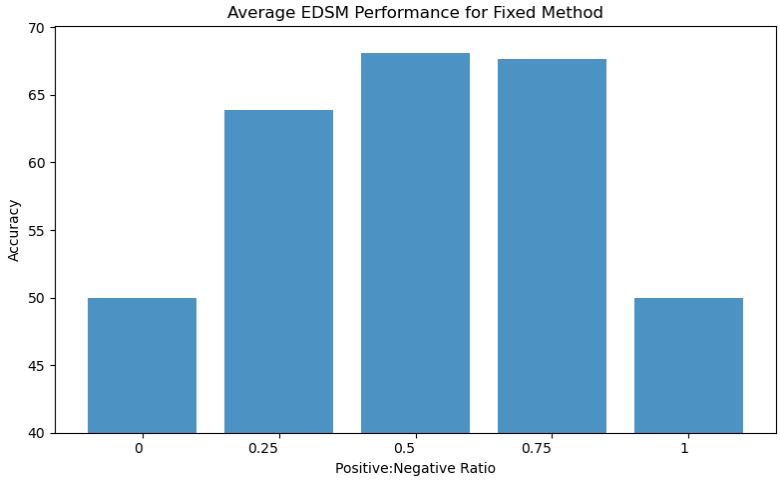


Figure 5.7: Mean accuracy for the EDSM using different ratios with fixed termination probability. This graph shows that EDSM achieves the highest performance with a balanced train set, but even positive proportions of 0.75% still produce comparable results.

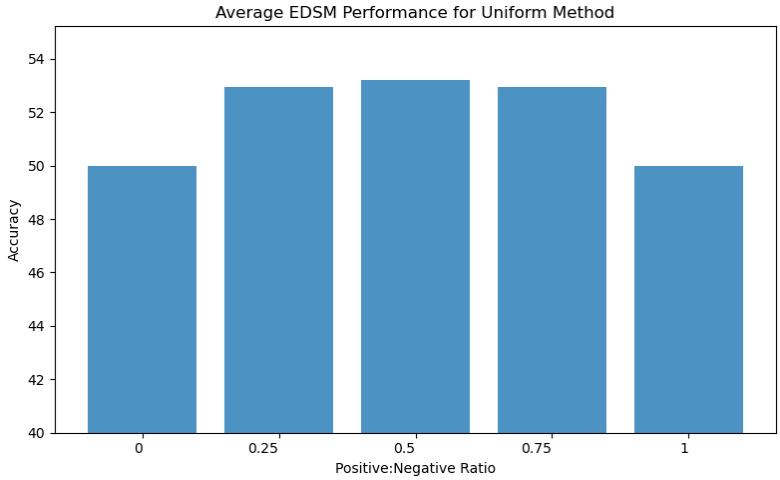


Figure 5.8: Mean accuracy for the EDSM using different ratios with uniform sampling. The performance is higher for the central columns, but the achieved accuracies are still extremely poor considering the training set size.

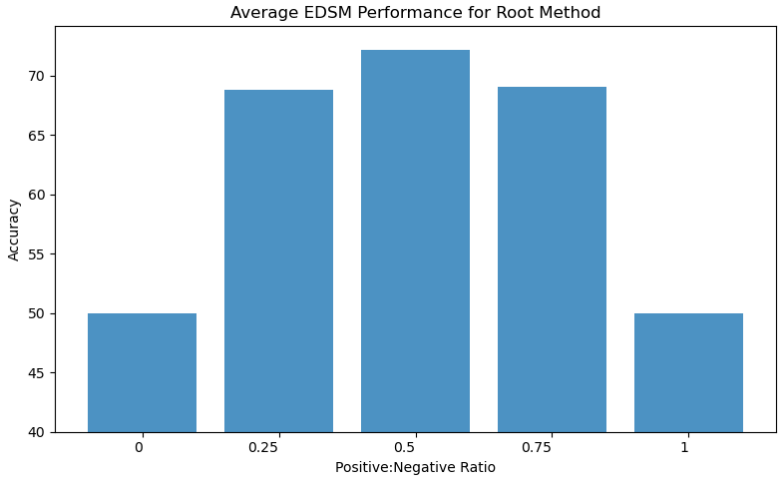


Figure 5.9: Mean accuracy for the EDSM using different ratios with root decay termination probability. In this experiment, the root walk outperformed the fixed termination walk significantly(p-value $1 * 10^{-3} << 0.05$). It achieved the highest mean accuracy around 73% for a positive ratio of 0.5

5.10 represents this as a box plot, summarizing the results over 100 different seeds. Despite the common intuition that bigger DFA should be harder to learn, the more complex structures have a marginally higher median accuracy. The explanation might be that the generated test cases are insufficient for covering the larger structures. Since the results from this experiment seem to imply little to no correlation between automata size and accuracy, we perform a statistical correlation test and thus derive a Pearson correlation value of 0.007 and a p-value of 0.78($>>$ 0.05). Under these experiment parameters, there is no significant dependency between recovered accuracy and CDFA size when using a database equivalence oracle to train an L^* algorithm.

Table 5.1: Average Accuracy of L* by DFA Size over all the different equivalence oracle sizes. We can see that the target's dimensions hardly influence the mean.

DFA Size	Average Accuracy (%)
(2,30)	96.995
(2,50)	96.619
(3,50)	97.444
(3,100)	97.128

5.1.5 Experiment 5

Before presenting the results, we describe some modifications to the experimental setup. Although we originally planned to include the classic RPNI implementation across all automaton sizes, excessive runtimes limit its evaluation to $|Q| = 25$. As shown in Table 5.2, EDSM requires, on average, 69 times longer to converge than its GSM variant. Extrapolating this trend linearly suggests an average runtime of approximately 52 785 seconds for $|Q| = 100$. Furthermore, GSM RPNI is

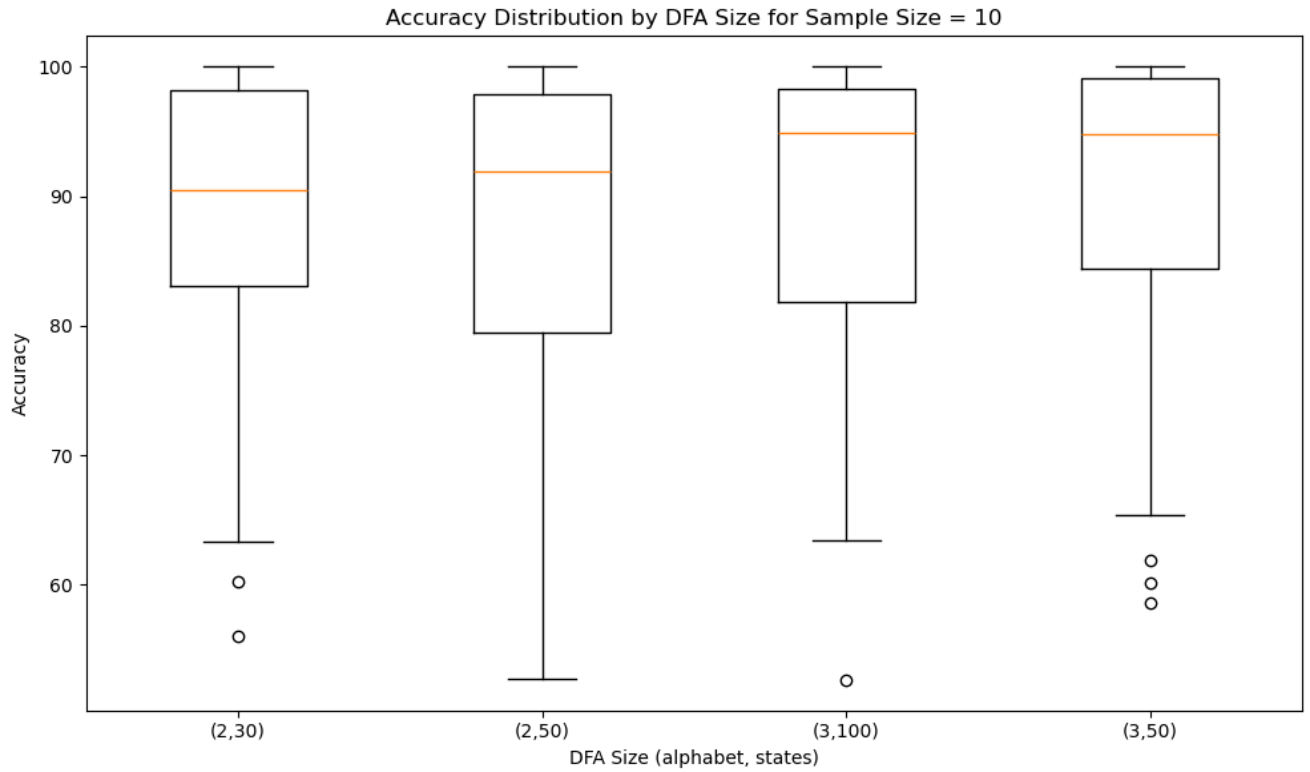


Figure 5.10: Accuracies achieved with the database-bound L* equivalence oracle (of size 10) for different automata sizes. The box plots show the distributions in a clear manner, showing that all of them have a similar shape and that the median is always above 90%.

more than 2000 times faster (on average) than the classical algorithm, justifying its adoption in all other experiments.

The L^* learner consistently achieves the shortest runtimes.

We consider the resulting leaning durations for both EDSM and RPNI-GSM to be generally reasonable for practical applications, especially if multiple tasks execute in parallel. Of course, given a convenient interface with the target and if the application allows, L^* is the go-to algorithm.

The performance for all learners exhibits superlinear runtime increases when moving from 25 to 50 states, which then transitions to a sublinear runtime in the range of 50 to 100 (see Table 5.2). The reason for this behavior is most likely that we use the fixed termination random walk, which does not explicitly adapt to the different automata sizes. Thus, the 100-state CDFA gets underexplored and, since runtime largely depends on the length of the inputs (and their amount), the growth in runtime is slower. However, larger structures are still harder to learn due to greater uniqueness in the training sample, leading to more repeated merge attempts. To explore this further, we conduct an additional follow-up experiment in Section 5.1.6.

Table 5.2: Mean runtimes in seconds by DFA Size and Algorithm using fixed random termination sampling. Some entries for RPNI Classic are missing due to extreme run times. It is important to notice the sublinear growth of runtime, after $\|Q\| = 50$ is reached.

Learners	$\|Q\| = 25$	$\|Q\| = 50$	$\|Q\| = 100$
EDSM	75.869	765.407	1588.980
RPNI Classic	5233.970	—	—
RPNI GSM	2.340	8.749	9.811
L*	0.017	0.074	0.080

5.1.6 Experiment 5.5

We incorporate three changes to the previous setup to increase the reliability and speed up the learning process. The first one is a reduction of the train set size by half (to 2500), followed by a simplification of the target languages by changing $|\Sigma|$ to 2. To avoid the issue encountered before, we use uniform sampling to obtain appropriate string lengths for each CDFA size. For that, the formula discussed in Section 3.1 ($2\log_2 n + 3$) is employed. Thus, the corrected code returns the following results (Table 5.3).

Again, one can see that EDSM is tens of times slower than RPNI-GSM and that L^* takes

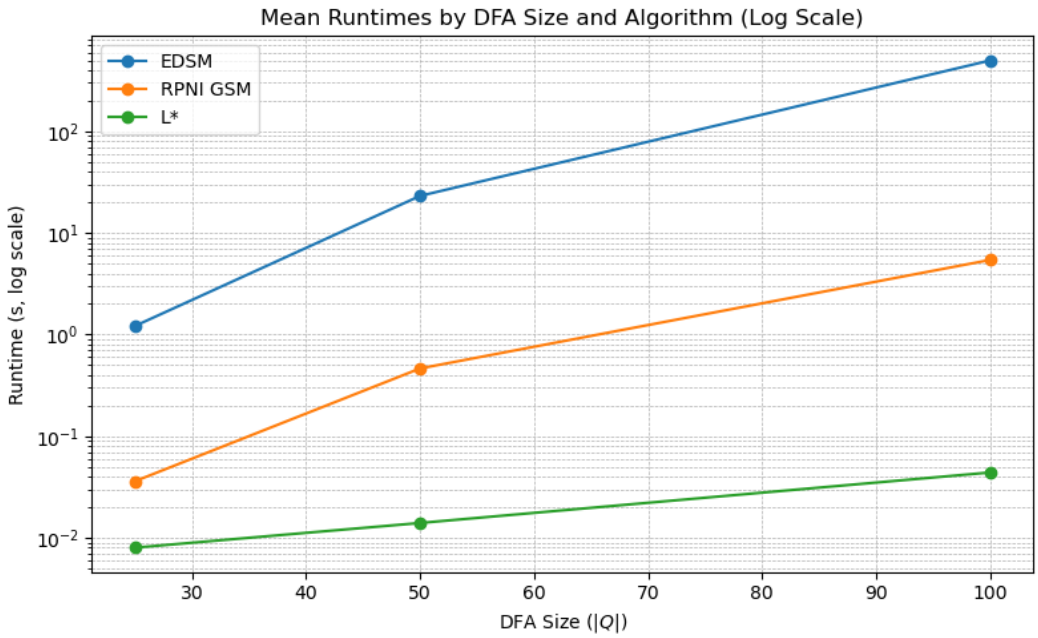
Table 5.3: Mean Runtimes by DFA Size and Algorithm using uniform sampling. We notice that halving the train set and the alphabet has noticeably reduced the overall runtimes.

Learner	$\|Q\| = 25$	$\|Q\| = 50$	$\|Q\| = 100$
EDSM	1.214	23.158	499.092
RPNI GSM	0.036	0.464	5.436
L*	0.008	0.014	0.044

less than a tenth of a second to reproduce the target. Most importantly, note that the exponential

growth in runtime indeed continues for more complex structures on the logarithmic graph in Figure 5.11.

Figure 5.11: Mean runtime as a function of DFA size for RPNI, EDSM, and L* on uniformly sampled traces (y-axis log scale). The roughly straight trajectories on the log-scale plot indicate that runtime grows super-linearly with the number of states.



5.1.7 Experiment 6

Table 5.4 shows the average accuracy of RPNI and EDSM algorithms across 100 runs for three different sampling strategies. The observed differences between methods are minimal - all results fall within a narrow range of around 52-54%. In general, that suggests that the choice of sampling method, whether using the **OpenFST** data loaded from a file or the previously used fixed-random termination and uniform sampling, has no statistically meaningful impact on the final accuracy of the learned automaton (in this setting). The only exception is that the mean RPNI score for the file method is significantly higher than the uniform one (p-value of $0.026 < 0.5$). All other p-values (for each possible pair, for the same learner) are bigger than 0.10. Moreover, the performance of EDSM and RPNI is nearly identical across all categories ($p = 0.59 > 0.05$), which is most likely due to the limited training set size, as concluded in Experiment 1 (5.1.1).

5.2 Addressing the RQs

In this section, we summarize the results by answering the research questions defined in Chapter 1:

Table 5.4: Mean Accuracy by Method and Algorithm. The random generation methods seem to perform similarly for small training sizes.

Method	RPNI Accuracy (%)	EDSM Accuracy (%)
File(OpenFST)	53.498	53.151
Uniform	52.419	52.879
Fixed Random	53.389	53.815

RQ_1: What algorithms for string generation provide the most diverse datasets?
We test many new ideas for string dataset generation algorithms, but most do not show remarkable results. An exception to that is the most memory-efficient method - the simple path explorer, which successfully accelerates the inference process of conventional passive learning techniques. The main limitation is that such a heuristic cannot entirely reconstruct the target DFA when relying on paths alone. As mentioned, a potential solution complements the simple path approach with randomly generated strings. In the majority of cases, none of the newly proposed probability decay functions outperform the fixed-termination one.

RQ_2: How do different automata learning algorithms compare in terms of accuracy and training efficiency?
In this thesis, we test 3 different learners, namely the passive RPNI and EDSM, and different variations of the active L^*. As expected, the more advanced EDSM outperforms RPNI in almost every scenario by a few %. In the cases where RPNI functions better (small datasets), the advantage it obtains is not statistically significant. L^*, as expected, proves to be outperforming both passive techniques. The attempts to essentially "handicap" the algorithm only prevent convergence to the ground truth, but it still retains an exceptional accuracy of nearly 100%. We notice the same trend in training efficiency: The active method finishes the learning process in less than a tenth of a second compared to the tens of minutes that classical passive implementations take. An exception to that is the GSM variation of RPNI, which is just 100 times slower than $L*$ on average (for larger DFA).

RQ_3: To what degree do the benchmarking results from previous research align with the results obtained in this project?
While we do not directly replicate specific experiments from previous studies, and most implemented components (such as learning packages, benchmark dataset generation, and, most importantly, target language initialization) differ, the results still broadly align with prior research.

Accuracy scores for RPNI and EDSM typically fall within the 50%–70% range, which aligns with findings reported in the literature [18]. Much of the related work focuses on alternative passive learning methods [2, 15] or applies the techniques in entirely different domains [5], which made direct comparisons challenging.

Nonetheless, as discussed in $\mathbf{RQ}_2$, the outcomes we observe in this study align with the theoretical expectations outlined in previous research. Finally, it is worth noting that in Experiment 6 (5.1.7), we show that our benchmark methods stay

consistent with widely used tools such as **OpenFST** at smaller scales.

Chapter 6

Conclusion

This chapter is dedicated to a reflection on our work and a few directions for future research possibilities.

This thesis sets out to expand the research on passive automata learning by suggesting new approaches to sampling languages with adequate coverage of an SUL. It investigates new generation techniques as well as different proportions of accepted to rejected strings, aiming to enhance dataset quality.

Our most notable contribution is the implementation of the simple-path-based heuristic. Unlike pure random sampling, this method explores the target DFA's structure, by ensuring that every state and transition is covered at least once. The results show that this approach significantly improves accuracy scores in early learning stages. In practice, such simple-path generation is preferable when high coverage is needed with limited data. For example, this would be most useful when exploring systems in which trace collection is expensive or constrained. In contrast, uniform sampling and fixed-termination random walks remain a solid baseline when there is no issue generating massive amounts of strings. We compare the different generation techniques on various automata sizes, establishing a clearer picture of where each method succeeds or struggles.

Experiments also show the limitations of passive learning, particularly in scenarios involving sparse datasets. Furthermore, all learners are evaluated based on their training speeds.

Lastly, we draw comparisons between the proposed generation methods and current automata benchmark tools.

6.1 Limitations

Early commitment to unfruitful ideas led to the somewhat limited scope of the research. As an example, most of the functions are compatible only with complete DFAs. Another issue is that most of the new methods do not perform well compared to established techniques. Experiments using uniform sampling could have been conducted using a more appropriate maximum length parameter.

Finally, the thesis does not cover the use of neural network approaches. Thus, newer methods are not evaluated.

6.2 Future Work

As noted in Section 5.1.1, a re-examination of how each decay-function parameter influences coverage and convergence is necessary. Fine-tuning termination probabilities and walk depth could lead to more efficient configurations. In addition, adjusting the positive-to-negative string ratio for each learning algorithm may reveal optimal sampling strategies.

The simple-path generator shows strong potential. Still, we can help it converge to its target by integrating it with random string selection. This hybrid strategy could provide both broader structural coverage and greater sample diversity. Results (5.1.2) indicate that such an approach may outperform individual generation methods.

The comparison to **OpenFST** in Experiment 6 (5.1.7) needs expansion to bigger train data and more complex automata to acquire more insightful results.

Expanding the benchmark to include neural network–based learners is another potential direction. Evaluating models such as LSTMs, GRUs, and Transformers on these datasets will offer insight into how well modern sequence learners compare to the classic passive approaches.

Lastly, updating the generation framework to support different types of structures, including stochastic DFAs, probabilistic automata, Mealy machines, and nondeterministic finite automata, will increase the scope and usability of our benchmark. Adapting current coverage-based oriented generation (such as the simple path heuristic) to these models introduces new challenges but provides valuable opportunities for broader applications.

Bibliography

[1] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the abbadingo one dfa learning competition and a new evidence-driven state-merging algorithm," in *Grammatical Inference: 4th International Colloquium (ICGI 1998), Proceedings*, ser. Lecture Notes in Computer Science, V. Honavar and G. Slutzki, Eds. Springer, 1998, vol. 1433, pp. 1–12.

[2] S. van der Poel, D. Lambert, K. Kostyszyn, T. Gao, R. Verma, D. Andersen, J. Chau, E. Peterson, C. St. Clair, P. Fodor, C. Shibata, and J. Heinz, "Mlregtest: A benchmark for the machine learning of regular languages," *arXiv preprint arXiv:2304.07687*, 2023. [Online]. Available: https://arxiv.org/abs/2304.07687

[3] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[4] J. Oncina and P. Garcıa, "Inferring regular languages in polynomial update time," in *Proceedings of the 4th IAPR Workshop on Applications of Formal Grammars in Pattern Recognition and in Biological Sequence Analysis*, 1992, pp. 49–60.

[5] B. K. Aichernig, E. Muškardin, and A. Pferscher, "Active vs. passive: A comparison of automata learning paradigms for network protocols," *arXiv preprint arXiv:2209.14031*, 2022.

[6] E. Muškardin, T. Burgstaller, M. Tappler, and B. K. Aichernig, "Active model learning of git version control system," in *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2024, pp. 78–82.

[7] K. Kanellopoulou, "Active automata learning for network protocols," Master's thesis, Graz University of Technology, Graz, Austria, Apr. 2016.

[8] D. B. Searls, "The computational linguistics of DNA," in *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, 1993, pp. 65–72.

[9] Y. Sakakibara, "Grammatical inference in bioinformatics," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 7, pp. 1051–1062, 2005.

[10] C. W. Omlin and C. L. Giles, "Extraction of rules from discrete-time recurrent neural networks," in *Neural Networks*, vol. 8, no. 2, 1996, pp. 273–284.

[11] K. Basye, "An automata-based approach to robotic map learning," in *AAAI Fall Symposium on Planning with Uncertainty and Learning and Motor Control*, ser. FS-92-02, 1992, pp. 1–6.

[12] B. Araki, K. Vodrahalli, T. Leech, C. Vasile, M. Donahue, and D. Rus, "Learning and planning with logical automata," *Autonomous Robots*, vol. 45, no. 7, pp. 1013–1028, 2021.

[13] A. Clark, "Unsupervised induction of stochastic context-free grammars using distributional clustering," in *Proceedings of the 2001 Workshop on Computational Natural Language Learning*, 2001.

[14] E. Rich, *Automata, Computability and Complexity: Theory and Applications*, University of Texas at Austin, January 2007, draft. Available at https://www.cs.utexas.edu/~ear/cs341/automatabook/AutomataTheoryBook.pdf.

[15] B. Lambeau, C. Damas, and P. Dupont, "State-merging dfa induction algorithms with mandatory merge constraints," in *Grammatical Inference: Algorithms and Applications. 9th International Colloquium, ICGI 2008, Saint-Malo, France, September 22–24, 2008. Proceedings*, ser. Lecture Notes in Computer Science, A. Clark, F. Coste, and L. Miclet, Eds. Berlin, Heidelberg: Springer, 2008, vol. 5278, pp. 139–153.

[16] C. Tirnauca, "A survey of state merging strategies for dfa identification in the limit," *Triangle: llenguatge, literatura, computacio*, no. 8, pp. 121–136, juny 2021. [Online]. Available: https://raco.cat/index.php/triangle/article/view/388851

[17] R. Weiss, V. Srikumar, and M. Lapata, "Extracting automata from recurrent neural networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2019.

[18] A. Soubki and J. Heinz, "Benchmarking state-merging algorithms for learning regular languages," in *Proceedings of the 16th International Colloquium on Grammatical Inference*, ser. Proceedings of Machine Learning Research, F. Coste, F. Ouardi, and G. Rabusseau, Eds., vol. 217. PMLR, Jul 2023, pp. 181–198, available at https://proceedings.mlr.press/v217/soubki23a.html. [Online]. Available: https://proceedings.mlr.press/v217/soubki23a/soubki23a.pdf

[19] Graphviz Development Team, *DOT Language*, Graphviz, 2025, accessed: May 28, 2025. [Online]. Available: https://graphviz.org/doc/info/lang.html

[20] M. Vazquez-Chanlatte, "dfa: Python library for modeling dfas, moore machines, and transition systems," https://pypi.org/project/dfa/, May 2024, version 4.7.1, released May 10, 2024; MIT License.

[21] DES-Lab, "AALpy: An automata learning library written in python," https://github.com/DES-Lab/AALpy, 2025, version 1.5.0, released February 28, 2025; MIT License.

[22] CS/ECE 374 Course Notes, "Note 1: How to convert dfa/nfa to a regular expression," University of Illinois at Urbana-Champaign, Department of Computer Science, Course Notes Version 1.0, Feb. 2019, spring 2019. Available at https://courses.grainger.illinois.edu/cs374/sp2019/notes/01_nfa_to_reg.pdf. [Online]. Available: https://courses.grainger.illinois.edu/cs374/sp2019/notes/01_nfa_to_reg.pdf

[23] OpenFst Development Team, "Openfst library," Google Research, 2024, https://www.openfst.org/twiki/bin/view/FST/WebHome.

[24] K. Gorman, "Pynini: A finite-state text processing library for python," https://pypi.org/project/pynini/, 2024, accessed 2025-06-30.