Faculty of Natural and Mathematical Sciences
Department of Informatics

King's College London,
Strand Campus, London,
United Kingdom

KING'S College LONDON

7CCSMPRJ
Individual Project Submission 2023/24

Name: Aishwarya Senthil Kumar
Student Number: 23131563
Degree Programme: Artificial Intelligence
Project Title: Implementation of L* learning algorithm (Rivest-Schapire variant)
Supervisor: Dr. Hana Chockler
Word count: 13567

☒ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature:

[illegible]

Date: 04 / 08 / 2024

Department of Informatics
King’s College London
United Kingdom

7CCSMPRJ MSc Project

IMPLEMENTATION OF L LEARNING ALGORITHM-(RIVEST-SCHAPIRE VARIANT)*

Name: Aishwarya Senthil Kumar
Student Number: 23131563
Degree Programme: Artificial Intelligence

Supervisor’s Name: Dr. Hana Chockler

This dissertation is submitted for the degree of MSc in Artificial Intelligence.

ACKNOWLEDGEMENT

I would like to thank the Almighty for blessing me with enough skill and competency, and providing me opportunities to showcase them.

I would like to thank my parents, Mrs. R. Vidya Kumar and Dr. R. S. Kumar, my brother Mr. Siddarth Senthil Kumar, for their continuous support and love.

I would like to thank my grandmother, Mrs. A. R. Saroja, for being a guardian angel in all of my endeavours.

I would like to thank my friend, Mrs. Regithra Surendhar, for being a beacon of light during the darkest times.

I would also like to express my gratitude to King’s College London for providing me with an opportunity and also, learn and interact with bright minds. I would also like to thank my guide, Dr. Hana Chockler, for the right guidance.

Last, but never the least, I would like to thank my husband, Mr. Clinton Michael, for his un wavering support, guidance and love, without whom, none of this would have been a possibility.

ABSTRACT

This project involves implementing the L* learning Algorithm proposed by Dana Angluin and also implementing the improvement of L* proposed by Ronald L. Rivest and Robert E. Schapire using Python.

In this project, it is proved how the improvement proposed is better than the L* proposed by Dana Angluin. It was proved that the number of membership queries is lesser and the number of rows in the observation table has significantly reduced.

Also, a simple robot environment, in the form of a DFA, is simulated using Simulink. From the environment, we could get the result of a membership query. We could understand from the simulated environment, if according to the proposed movement, the robot will be in its final position or not.

NOMENCLATURE

DFA	Deterministic Finite Automaton
λ	Epsilon[1]
L	Language[1]
A	Alphabet[1]
Q	Set of all states[1]
q_0	Initial state[1]
F	Set of final states[1]
S	Prefix-closed set of strings, [1]
E	Suffix-closed set of strings[1]
$\in$	Belongs to [1]
$\mathcal{E}$	Actual DFA[1]
$\mathcal{E}'$	Learned DFA[1]

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Aims and Objectives [1]

Automata learning is a specialized field within computer science focused on the construction of finite-state models, particularly deterministic finite automata (DFAs), from observed input-output data. This field is crucial for applications that require understanding and modelling the behaviour of systems based on their external interactions. By learning an automaton from examples, one can gain insights into the underlying structure and rules governing a system's behaviour, which is valuable for tasks such as robot environment modelling, software verification, protocol analysis, and natural language processing.

In this thesis, we delve into two prominent automata learning algorithms: Angluin's L* algorithm and the Rivest-Schapire improvement on L*. Both algorithms aim to learn a DFA, thereby, trying to model a deterministic environment.

By conducting a detailed empirical comparison, the thesis aims to highlight the practical benefits and trade-offs of the Rivest-Schapire improvement over Angluin's original L* algorithm. This comparison provides valuable insights into the performance and applicability of these algorithms in real-world scenarios where efficient and accurate automata learning is crucial.

In addition to comparing these algorithms, we create a robot environment where the robot gives the result of a membership query, thereby enables modelling of the environment. This practical setup serves as an illustrative example of how automata learning algorithms can be applied to real-world scenarios.

1.1.1 Dissertation Length

The dissertation consists of 13567 words.

1.2 Motivation

The motivation for this project is driven by the need to effectively understand and model the behaviour of complex systems through finite-state models, particularly deterministic finite automata (DFAs). Automata learning is a specialized field within computer science that addresses this need by constructing DFAs from observed input-output data.

DFAs are mathematical models used to represent systems with a finite number of states, which have deterministic transitions between those states based on inputs. They are particularly useful in robotics for the following reasons:

- **Predictable Behaviour:** DFAs provide a clear and predictable framework for modelling the robot's behaviour in response to specific inputs, which is essential for ensuring reliable performance.
- **Simplification**: By abstracting the complex interactions into a finite number of states and transitions, DFAs simplify the understanding and analysis of the system.
- **Verification and Validation:** DFAs facilitate formal verification and validation processes, ensuring that the robot's behaviour adheres to desired specifications and safety requirements.

1.3 General setup

The general setup of this project includes following key components and steps.

1.3.1 Problem Statement

The objective is to construct a DFA that accurately models a given system based on observed input-output data. The context is to utilize a robot interacting with its environment as a concrete example of the learning process.

1.3.2 Algorithms to be Compared

We will be comparing Angluin’s L* algorithm, a foundational algorithm in active automata learning that constructs a DFA through membership and equivalence queries. Also, we will be implementing the Rivest-Schapire improvement, which is an enhanced version of the L* algorithm that optimizes the processing of counterexamples to improve learning efficiency.

1.3.3 Robot Environment

The environment is deterministic, where the robot's actions serve as inputs to the automaton, and the resulting states are outputs. The robot performs actions (e.g., moving, turning) and observes the resulting changes in state. These actions and observations are used to construct the DFA.

1.3.4 Learning Process

The DFA uses Active Learning Framework. The robot interacts with the environment to gather information and refine its model. The learning process involves two types of queries:

- **Membership Queries:** The robot asks whether specific sequences of actions lead to particular states (e.g., "Does moving forward twice and turning left lead to state X?"). [1]
- **Equivalence Queries:** The robot proposes a hypothesized DFA and asks if it is equivalent to the target DFA. If not, a counterexample is provided. 1
- **Counterexample Handling:** When a counterexample is encountered, it is used to refine the DFA. The Rivest-Schapire improvement focuses on optimizing this process to reduce redundant queries and improve learning efficiency. [1]

1.3.5 Performance Metrics

- **Number of Membership Queries:** Total membership queries made during the learning process.
- **Number of Equivalence Queries:** Total equivalence queries made.
- **Membership Table Size:** Size of the observation table used to construct the DFA.

1.3.6 Empirical Evaluation

Conducted experiments to compare the performance of both algorithms in the robot environment. Analysed results to highlight the practical benefits and trade-offs in terms of query efficiency, learning speed, and computational resources.

2 Background and Literature Survey

2.1 Introduction

As depicted by Rivest and Schapire, the problem statement can be illustrated using the scenario of a robot trying to learn about its environment. In this context, the environment is assumed to be deterministic, meaning that for a given state and action, the resulting state is always the same. The actions performed by the robot serve as inputs to an automaton, which models the environment's behaviour based on these inputs.

2.2 Active Learning Method

Rivest and Schapire adopted the active learning method originally developed by Angluin, and they proposed improvements to enhance its efficiency. In active learning, the learner (in this case, the robot) interacts with the environment (or a teacher representing the environment) to gather information and refine its understanding or model of the environment.

2.3 Learning Process and Counterexamples

In the learning process, the robot performs actions and observes the resulting states. These actions and observations are used to build and refine a deterministic finite automaton (DFA) that models the environment. A crucial part of this process involves handling counterexamples, which are discrepancies between the learned model and the actual behaviour of the environment.

A counterexample is given when the output state produced by the robot's actions in the environment does not match the output state predicted by the learned DFA. When such a discrepancy is found, it indicates that the current model is incorrect or incomplete. This helps in directing the robot to learn and improve its model.

2.4 Identifying Counterexamples

The robot identifies counterexamples through a process of random search. It performs a sequence of actions and compares the observed state transitions with those predicted by the learned DFA. If a mismatch is detected, the sequence of actions leading to the mismatch is considered a counterexample. This counterexample is then used to refine the DFA, improving its accuracy.

2.5 Improvement by Rivest and Schapire

Rivest and Schapire's key improvement over the original L* algorithm lies in how counterexamples are processed. Their method focuses on efficiently incorporating counterexamples into the learning process, which can significantly reduce the number of queries needed and the overall runtime. By optimizing counterexample processing, their algorithm aims to quickly correct the learned DFA and converge to the accurate model of the environment.

2.6 Evaluation of Algorithms

The extensive material used to understand the differences between the base L* algorithm and the Rivest-Schapire variant is detailed in a paper published by Rick ten Tije[1]. This paper provides an in-depth analysis of how counterexample processing can significantly alter runtime and the processing resources required for learning. Rick ten Tije's paper explores the following aspects.

2.6.1 Counterexample Processing

The paper details how efficiently handling counterexamples can lead to faster convergence of the learned DFA. By minimizing redundant queries and optimizing the update process, the improved algorithm can achieve better performance.

2.6.2 Runtime Analysis

The paper compares the runtime of both algorithms, showing how the Rivest-Schapire improvement can reduce the time required to learn the correct DFA. This is crucial in practical applications where learning efficiency directly impacts the system's performance.

2.6.3 Resource Utilization

The analysis includes the computational resources needed for both algorithms. Efficient counterexample processing not only reduces runtime but also lowers the computational overhead, making the learning process more resource-efficient.

3 Background Theories

3.1 Fundamentals of DFA [1]

An alphabet A is a finite set of letters, over which the words are defined. Combinations of these letters are called words.[1] All combinations of words obtained from A is denoted as A^*. In this context, any language L, can be denoted as a subset of A^* $(L \subseteq A^*)$.[1] The empty word or string denoted by ‘e’, λ or epsilon, is always a part of the language and therefore in A^*.[1]

A deterministic finite-state automaton also known as DFA, is a type of automaton that accepts only strings from a particular language, that is it either accepts, or rejects a given word. [1] A DFA is a 5-tuple $M= (Q, A, \delta, q_0, F)$ where Q is the set of states, A the alphabet, $\delta\colon Q \times A \rightarrow Q$ the transition function, $q_0 \in Q$ the initial state and $F \subseteq Q$ the set of final states.[1]

We can define the transition function for words δ^* in a DFA M as $Q \times A^* \rightarrow Q$ where Q is the set of states in M and A^* is the set of words.

In order to check if any particular word or a string belongs to a DFA or is accepted by a DFA, we use the transition function δ *. Finally, if the resulting state of this function is one the final states, i.e., in F, then we can say that the automaton accepts the given word or string.

As mentioned before, a finite automaton accepts a set of strings or a particular language. A language can be defined as follows. For any Finite Automaton, $M = (Q, A, \delta, q_0, F)$, the language of state $q \in Q$ is defined as: $L_q = \{w \mid \delta_* (q_0, w) \in F\}$. The language accepted by M, i.e., the automaton is the language of the initial state L_{q0}.

3.2 L* Learning Algorithm [2]

In this section, the L* learning algorithm, used for learning DFAs will be discussed in detail. This algorithm was proposed by Dana Angluin and uses queries and counterexamples. Assume that we have an environment to be modelled as DFA. The DFA consists accepts a language over a known alphabet A. Our objective is to find the DFA which represents the Language and in turn the environment in the form of a DFA.

3.2.1 Minimally Adequate Teacher

In L*, we have a teacher termed as Minimally Adequate Teacher, the language L and also the DFA is not known to us. But the Minimally Adequate Teacher is aware of the Language and the DFA.

3.2.2 Membership and Equivalence queries

In this algorithm, we ask the Minimally Adequate Teacher (shortly MAT) two queries.

- Membership Query: If any given word or string is in the language L.
- Equivalence Query: If the DFA learned is equivalent to the actual DFA.

The Minimally Adequate Teacher, in case of Membership queries, responds either as True or ‘1’ if the word is in the Language L or responds with ‘0’ or False if it is not. In case of the equivalence queries, the Minimally Adequate Teacher will give the output as True or ‘1’ if the DFA learned is the same as the actual DFA or it will give the output as False or ‘0’. This means that for equivalence queries, the Minimally Adequate Teacher, will either give an output with a yes, which means that the created automaton represents the Language L or with a no and a counterexample.

3.2.3 Counterexample

The counterexample distinguishes the learned DFA and the actual DFA. It means, that the counterexample is either accepted by the learned DFA but not accepted by the actual DFA, or the counter example is not accepted by the learned DFA, but not accepted by the actual DFA.

After we create a DFA using the algorithm, i.e., a hypothesis is created by the algorithm, we run the equivalence query. We ask the teacher if the created DFA is the same as the actual DFA.

If the Minimally Adequate Teacher responds with a ‘True’ or ‘1’, the algorithm terminates. But if the Minimally Adequate Teacher responds with a ‘False’ or ‘0’, then it will give a counterexample. After the counterexample is given, L* adds it and all of its prefixes to *S*. Now the algorithm continues to make a new table and a new DFA is generated. This process will continue until the learned DFA is the same as the actual DFA, that is the generated automaton represents the required language L.

3.2.4 Prefix and Suffix

The algorithm uses observation tables to create the automaton or the DFA[1]. In order to define the automaton, we have to understand what is meant by prefix and suffix. [1]

The starting part of any word is called a prefix. i.e., a set of letters *p* is a prefix of word *w* if there is a set of letters *u* such that $pu = w$. For any word or a set of letters, a prefix is a set of all sub-words of that word which start at the beginning of the word. [1]

A set of letters *s* is a suffix of any sequence of characters *w*, if there is a set of characters *u* such that $us = w$.

3.2.5 Prefix and Suffix closed

We may call a particular set of strings as Prefix-closed if every prefix of each and every member of the set is also a member of the set, [1] i.e., every member of the set is a prefix to at least one member. Similarly for suffix-closed, every suffix of each and every member of the set is also a member of the set, [1] i.e., every member of the set is a suffix to at least one member.

3.2.6 Observation Tables [1]

An observation table or a membership table consists of three elements *(S, E ,row)* where

- *S* is a prefix-closed set of words, [1]
- *E* is a suffix-closed set of words[1]
- row a function defined as row: $S \cup S \times A \rightarrow \{1, 0\}$ E. [1]

In any observation table, also called as membership table, the labels of the rows in upper table is S and labels of the rows in lower table represents all the words in the Cartesian product of S and A which are not already in S, and the column labels denote E.

Each value in the table corresponds to a word which consists of a concatenation of the row label and the column label. The row now maps the word to either a '1' or a '0' depending on the fact if the word is in the unknown language L. Any given word is determined if they are present in the Language or not by the membership query, The Minimally adequate teacher responds with a Yes or No.

3.2.7 Consistency and Closedness [1]

The algorithm can derive the DFA from any observation table when it has the following properties.

i. Consistency [1]: Let us consider two set of strings $s_1, s_2 \in S$ such that row(s1) = row(s2), then for all a in A it must hold that row (s1 · a) = row (s2 · a).
ii. Closedness [1] : It means that for each row in *S*, there should be an equal row in S.A. Therefore, for every *s1* $\in S \cdot$ A there exists an *s2* $\in S$ such that *row (s1) = row (s2).*

Closedness is necessary because if any row in *S.A* is not in *S*, then that row is not converted into a state and thus impossible to transition to. [1]

Consistency is also necessary because, without it, unique transition to each and every state would become impossible, thereby breaking the idea of DFA.[1]

3.2.8 Hypothesis construction

The DFA can be constructed only from a closed and consistent observation table. The learned DFA or the constructed DFA constructed from the observation table or membership table is called the Hypothesis.

For any observation table *O = (S, E, row)* that is closed and consistent, a hypothesis *M = (Q, A, δ, q0, F)* can be created where: [1]

- Q = {r | r ∈ S} [1].
- *A* is the alphabet [1].
- δ(row(s), a) = q ∈ Q where row (s · a) ∈ q, for s ∈ S and a ∈ A. [1]
- q_0 = q ∈ Q where row(λ) ∈ q. [1]
- F = {q ∈ Q | r(λ) = 1 for r ∈ q}. [1]

3.2.9 Overview of L* Learning algorithm[1][2]

i. The algorithm begins by initializing the sets *S* and *c* with the empty word, denoted by *epsilon*.
ii. Next, the algorithm constructs an observation table using *S, E* ,the alphabet *A* , and the rows corresponding to *S*.
iii. The L* algorithm then checks the observation table for closedness and consistency. If the table is not consistent, it identifies the set of letters causing the inconsistency and adds them to the set *E*. The observation table is then updated accordingly.
iv. If the observation table is not closed, the algorithm adds the word $s \cdot a$ (where $s \in S$ and $a \in A$) to *S* if *row(s·a)* is not already in *S*, and updates the table.

v. After addressing any closedness and consistency violations, the algorithm repeats the process to ensure that the observation table remains closed and consistent.

vi. Once the observation table is confirmed to be closed and consistent, the algorithm constructs the learned DFA (hypothesis) and presents it to the teacher. The teacher then responds with either 'True' or 'False'.

vii. If the Minimally Adequate Teacher responds with 'False', the counterexample provided by the teacher and its prefixes are added to *S*. The algorithm then restarts the process, checking for closedness and consistency violations again.

viii. If the teacher responds with 'True', the hypothesis correctly represents the unknown language, and the algorithm terminates.

3.3 Rivest-Schapire Improvement [3][2][1]

The following block of text was referred from the paper published by Rivest and Schapire[3] and also the paper published by Rick en Tije. [1] A variant of Angluin's L* algorithm proposed by Rivest and Schapire is described. The Rivest-Schapire variant improves the number of membership queries made by the inference procedure and also the number of rows in the observation table significantly.

As described previously, Angluin's algorithm contains an observation table in the form of (*S*, *E*, *T*). The observation table records the value *T(x)* for each string *x* belonging to (*S U SB*) *E*. (In this case, *q0* is the initial state in Angluin's model, to which the automaton can always be reset.)

The data of *T* will be inserted using membership queries, and the number of queries is the cardinality of (*S U SB) E*. In Angluin's algorithm, |*S*| is bounded by *O(mn)* and |*E*| by *O(n)*. This improvement ensures that |*S*| is bounded by *O(n)*. To achieve this on |*S*|, *n log m* queries are essential, and hence the overall bound on the number of membership queries is *O(kn^2 + n log m)*.[3]

As described in the previous sections, *S* is a prefix-closed set of strings. Unlike the L* algorithm described by Angluin, in this variant, we maintain a condition that for all *s1* and *s2* belonging to S, if $s1 \neq s2$ then $q0\ s1 \neq q0\ s2$. Therefore, $|S| \leq n$ is maintained at all times. Also, *S* only grows in size (strings are never deleted from *S*). The set *E* represents a set of experiments that distinguish the states of *S* (i.e., the states *q0 s* for *s* belonging to *S*).

This algorithm is very similar to Angluin's L* algorithm. Initially, *S* and *E* are initialized to the set $\{\lambda\}$. Using membership queries, fill in the entries of table *T* and make *(S, E, T)* closed. Then, from *(S, E, T)*, construct and conjecture machine $\mathcal{E}'$. If the conjecture is correct, quit. Otherwise, update the set *E* using the returned counterexample, and repeat until a correct conjecture is made.

We say observation table *(S, E, T)* is closed if for all *s* belonging to *SB* there exists *s'* belonging to *S* such that *row(s) = row(s')*. We know that *row(s)* is that function $f: E \rightarrow \{0, 1\}$ for which *f(e) = T(se)*. If *s* belonging to *SB* witnesses that *(S, E, T)* is not closed, then *s* is simply added to *S* (and *T* updated using membership queries). Note that this maintains the condition that all rows of *S* are distinct (and thus, the states to which they lead from *q0* are also distinct).

As we have seen earlier, Angluin's algorithm also requires that the observation table be consistent, that is, that *row(s1, b) = row(s2, b)* whenever *row(s1) = row(s2), s1, s2* belonging to *S* and *b* belonging to *B*. However, since our algorithm maintains the condition that $row(s1) \neq row(s2)$ for $s1 \neq s2$, this condition is always trivially satisfied.

Finally, if $\mathcal{E}'$ is different from $\mathcal{E}$, a counterexample *z* is obtained, and the set *E* must be updated. Our algorithm adds only a single string to *E* using *z*. However, to find this string, the procedure makes up to $log\ |z|$ membership queries.

The key property that must be satisfied by the new experiment *e* (which will be added to *E*) is the following: for some *s, s'* belonging to *S* and *b* belonging to *B* for which *row(s) = row(s'b)*, it must be that the experiment *e* must witness that *q0s* and *q0s'b* are different states. If this property is satisfied, then adding *e* to *E* will cause *S* to increase by at least one (to maintain closure).

We can now see how the counterexample processing happens. For $0 \leq i \leq |z|$, let $p_i\ r_i$ be such that $z = p_i\ r_i$ and $|p| = i$. $s_i = \delta'(\lambda, p)$ is the state reached after the first *i* symbols of *z* have been executed. Recall that *s* is both the state of $\mathcal{E}'$, and a string of *A*.

On input $\mathcal{E}$, the machine reaches a state outputting a particular value, which is not the same as the output given by the machine learned by us $\mathcal{E}'$. That is, if the output is 0 by the actual DFA, then the output is 1 by the learned DFA.

The counterexample *z* is split into two parts. The second part of *z, u,* was put through the hypothesis to obtain the prefix of the state in which *u* ended. If the hypothesis is a DFA, we will always end up in the same state when putting the same word through the DFA and thus we would obtain a single prefix representing the end state. That is done using the binary search. The remaining half of the string is added as a column to the set *E*.

3.4 Conclusion

The background theories discussed provide a comprehensive understanding of deterministic finite-state automata (DFA), the L* learning algorithm, and the Rivest-Schapire improvement.

Fundamentals of DFA:

A DFA is defined by a 5-tuple and accepts or rejects strings from a language. The language of a DFA is determined by its transition function and the set of final states. This fundamental understanding of DFAs forms the basis for automaton theory and computational language processing.

L* Learning Algorithm:

Proposed by Dana Angluin, the L* algorithm uses a minimally adequate teacher (MAT) to learn DFAs. The process involves membership and equivalence queries, with counterexamples guiding the refinement of the hypothesized DFA. The algorithm constructs and updates observation tables, ensuring they are prefix-closed, suffix-closed, consistent, and closed. The learning process continues until the hypothesized DFA matches the actual DFA, efficiently modeling the unknown language.

Rivest-Schapire Improvement:

This variant of the L* algorithm optimizes the number of membership queries and reduces the size of the observation table. By ensuring unique transitions and maintaining a bounded size for the set S, the algorithm enhances efficiency. The counterexample processing is streamlined, using binary search to update the set of experiments E with minimal queries. This improvement significantly reduces the computational complexity of learning DFAs.

Together, these theories illustrate the progression from the basic principles of DFAs to advanced learning algorithms, highlighting the importance of efficiency and accuracy in automaton learning and language recognition.

4 REQUIREMENTS AND SPECIFICATIONS

4.1 Requirements and specifications of L* by Angluin

4.1.1 Equivalence Query Function

Requirement: Check if the learned DFA is equivalent to the actual DFA.

Specifications:

- Input: `Learned DFA`
- Output: `True` if `Learned DFA` matches `ACTUALDFA`, otherwise `False`.

4.1.2 Membership Query Function

Requirement: Determine if an input string is accepted by the DFA.

Specifications:

- Input: `String` (a string to test)
- Output: `1` if the string is accepted by the DFA, `0` if rejected or contains invalid characters.

4.1.3 Alphabet Check Function

Requirement: Verify if all characters in a string belong to a specified alphabet.

Specifications:

- Input: `String` (a string to check), `ALPHABET` (list of allowed characters)
- Output: `True` if all characters are in the alphabet, otherwise `False`.

4.1.4 Initialize Function

Requirement: Initialize the observation table.

Specifications:
- Output: Two dataframes representing the observation tables are created.

4.1.5 Build Observation Table Function

Requirement: Populate the observation table using membership queries.

Specifications:
- Input: Two dataframes (Table1, Table2)
- Output: Updated dataframes with values filled.

4.1.6 Check Closedness Function

Requirement: Check if the observation table is closed.

Specifications:
- Input: Two dataframes (Table1, Table2)
- Output: \`True\` if \`Table2\` is a subset of \`Table1\`, otherwise \`False\`.

4.1.7 Column Variable Function

Requirement: Find a variable to add as a new column when necessary.

Specifications:
- Input: Two dataframes (Table1, Table2)
- Output: The name of the new column variable or \`None\` if no new column is needed.

4.1.8 Check Consistency Function

Requirement: Check if the observation table is consistent.

Specifications:
- Input: Two dataframes (Table1, Table2)
- Output: \`True\` if the tables are consistent, otherwise \`False\`.

4.1.9 Add Row Function

Requirement: Add a row to the observation table.

Specifications:

- Input: Two dataframes (Table1, Table2)
- Output: Updated dataframes with a new row added.

4.1.10 Add Column Function

Requirement: Add a column to the observation table.

Specifications:

- Input: Two dataframes (Table1, Table2)
- Output: Updated dataframes with a new column added.

4.1.11 Prefix Closed Function

Requirement: Create a prefix-closed set of strings from the given row headers.

Specifications:

- Input: `label` (list of row headers)
- Output: A list of prefix-closed strings.

4.1.12 Dataframe To Dfa Function

Requirement: Convert an observation table to a DFA.

Specifications:

- Input: Two dataframes (Table1, Table2)
- Output: A DFA constructed from the observation tables

4.1.13 Counter Example Function

Requirement: Add a counterexample to the observation table and update it accordingly.

Specifications:

- Input: Two dataframes (Table1, Table2)

- Output: Updated dataframes with counterexamples added.

4.2 Requirements and Specifications of L* improvement by Rivest-Schapire

4.2.1 Equivalence Query Function

Requirement: Determine if the learned DFA is equivalent to the actual DFA.

Specification:
- Input: A learned DFA
- Output:'True' if the learned DFA matches the actual DFA, `False` otherwise.

4.2.2 Membership Query Function

Requirement: Check if a given string is accepted by the DFA.

Specification:
- Input: A string.
- Output: `1` if the string is accepted by the DFA, `0` otherwise.

4.2.3 Alphabet Check

Requirement: Verify if all characters in the string are part of the DFA's alphabet.

Specification:
- Input: A string and a list of valid characters (alphabet).
- Output: `True` if all characters in the string are in the alphabet, `False` otherwise.

4.2.4 Initialize function

Requirement: Create and initialize two observation tables for the DFA.

Specification:
- Input: None.
- Output: Two empty DataFrames initialized with epsilon transitions.

4.2.5 Build Observation Table Function

Requirement: Populate the observation tables with the results of membership queries.

Specification:

- Input: Two DataFrames.
- Output: Updated DataFrames with results from membership queries.

4.2.6 Check Closedness Function

Requirement: Check if the second observation table is closed with respect to the first one.

Specification:

- Input: Two DataFrames.
- Output:`True` if the second table is closed under the first table, `False` otherwise.

4.2.7 Column Variable Function

Requirement: Determine which column variable (if any) needs to be added next to the observation tables.

Specification:

- Input: Two DataFrames.
- Output: The variable (if any) to add as a new column, `None` if no such variable is needed.

4.2.8 Check Consistency Function

Requirement: Check if the observation tables are consistent.

Specification:

- Input: Two Data Frames.
- Output: `True` if the tables are consistent, `False` otherwise.

4.2.9 Add Row Function

Requirement: Add a row to the top of the observation tables and update them.

Specification:
- Input: Two Data Frames.
- Output: Updated Data Frames with the new row added.

4.2.10 Add Column Function

Requirement: Add a column to the observation tables and update them.

Specification:
- Input: Two Data Frames.
- Output: Updated Data Frames with the new column added, or the original Data Frames if no column was added.

4.2.11 Prefix Closed Function

Requirement: Generate a prefix-closed set of strings.

Specification:
- Input: A list of strings.
- Output: A list of prefix-closed strings.

4.2.12 Data Frame To Dfa Function

Requirement: Construct a DFA from the observation tables.

Specification:
- Input: Two DataFrames.
- Output: The learned DFA as a list containing the transition function, initial state, final states, and states.

4.2.13 Output Sequence Function

Requirement: Generate the output sequence for a given state and sequence of inputs.

Specification:
- Input: A state and a sequence of inputs.
- Output: A string representing the output sequence.

4.2.14 Output State Function

Requirement: Compute the final state after processing a sequence of inputs from a given DFA.

Specification:
- Input: A DFA, an initial state, and a sequence of inputs.
- Output: The final state after processing the sequence.

4.2.15 Output Value Function

Requirement: Determine if a sequence is accepted by the given DFA.

Specification:
- Input: A DFA and a sequence.
- Output: `1` if the sequence ends in a state which is accepted, `0` otherwise.

4.2.16 Generate Counter Example Function

Requirement: Generate counterexamples for testing DFA equivalence.

Specification:
- Input: None.
- Output: Updates the global counterexample list with new strings.

4.2.17 Counter Example Function

Requirement: Find a counterexample that demonstrates the difference between the learned DFA and the actual DFA.

Specification:
- Input: A learned DFA.
- Output: A counterexample string.

4.2.18 Alpha Function

Requirement: Determine if the state after processing a given string is an accepting state.

Specification:

- Input: A DFA, a string, and an index.
- Output: `1` if the state is an accepting state, `0` otherwise.

4.2.19 Binary Search Function

Requirement: Perform a binary search on a string to find substrings that distinguish between two states.

Specification:

- Input: A DFA and a string.
- Output: A tuple of substrings and a distinguishing character.

4.2.20 Rivest Function

Requirement: Improve the DFA using the Rivest-Schapire algorithm.

Specification:

- Input: A learned DFA and two observation tables.
- Output: Updated observation tables.

4.2.21 Prefix closed Rivest Function

Requirement: Generate a prefix-closed set of strings for the Rivest-Schapire algorithm.

Specification:

- Input:A list of strings.
- Output: A prefix-closed set of strings.

4.2.22 Suffix Closed Rivest Function

Requirement: Generate a suffix-closed set of strings for the Rivest-Schapire algorithm.

Specification:

- Input: A list of strings.
- Output: A suffix-closed set of strings.

5 DESIGN

5.1 Design of the Angluin's L*

5.1.1 Overview

The goal of the code is to learn a DFA given a set of queries. The DFA is described by:

- A transition function
- An initial state
- A set of final (accepting) states
- A set of states

The code starts with an empty observation table and iteratively refines it until the learned DFA is equivalent to the target DFA.

Table1 variable in code represents the top part of the observation table. Table2 variable in code represents the bottom part of the observation table.

5.1.2 Constants

ACTUALTRANSITION	Defines the DFA transitions. This is a dictionary where each state maps to another dictionary that represents transitions for different input symbols.
ACTUALINITIAL	The initial state of the DFA.
ACTUALSTATES	List of states in the DFA.
ACTUALFINAL	List of final (accepting) states.
EPSILON	Represents the empty string.
ALPHABET	List of symbols that the DFA can process, including the epsilon symbol.
ACTUALDFA	A tuple containing the DFA transition function, initial state, final states, and all states.
COUNT	A counter for the number of membership queries made.
COUNTEREXAMPLE	Stores counterexamples encountered during the learning process.

Table 1: List of constants for Angluin's L* algorithm

5.1.3 Functions

Equivalence Query of Learned DFA: Checks if the learned DFA is equivalent to the actual DFA. Compares the learned DFA with the ACTUALDFA and returns True if they match, False otherwise.

Membership Query of a String: Determines if the given input string is accepted by the DFA. Updates the global COUNT variable with each query. Uses the DFA transition function to process the input string and checks if the resulting state is in the set of final states.

Alphabet Check: Verifies if all characters in the input string are within the specified alphabet. Returns True if valid, False otherwise.

Initialize: Creates and initializes the observation tables (Table1 and Table2). Calls Build Observation Table to populate these tables.

Build Observation Table of Table 1 and Table 2: It fills the observation tables with the results of membership queries. Calls Membership Query for each combination of row and column entries.

Check Closedness of Table 1 and Table 2: Checks if all rows in Table2 are present in Table1. Used to determine if the observation table is closed (i.e., complete for the current state space)

Column Variable: Identifies the next column to be added to the observation tables to ensure consistency. Returns a new column variable based on the current tables.

Check Consistency: Ensures that the observation table is consistent (i.e., rows that are equivalent should have the same responses to the current columns).

Add Row: Adds a new row to Table1 and Table2 by incorporating new prefix-closed strings. Updates the observation tables accordingly.

Add Column: Adds a new column to Table1 and Table2. Uses ColumnVariable to determine the new column and updates the tables.

Prefix Closed: Generates a prefix-closed set of strings based on the given row headers. Ensures that the set of strings includes all prefixes of existing strings in the table.

Dataframe To Dfa: Converts the observation tables into a DFA representation. Builds the transition function, initial state, final states, and state list from the tables.

Counter Example: Adds counterexamples to the observation tables to ensure that the learned DFA is equivalent to the actual DFA. Continually adds rows until the membership query for the new row is accepted.

Main Execution: Initializes the observation tables. Iteratively adds rows and columns to the tables until they are closed and consistent. Converts the tables into a DFA and checks equivalence with the actual DFA. Uses counterexamples to refine the tables if necessary.

5.1.4 Execution

Start with empty observation tables and fill them with membership query results. Refine the tables until they are both closed and consistent. Construct the DFA from the observation tables. Compare the learned DFA with the actual DFA. If not equivalent, use counterexamples to further refine the tables.

5.1.5 Output

The following are printed as an output.

- LearnedDfa: The DFA learned by the algorithm.
- COUNT: The number of membership queries made.
- Table1 and Table2: The final observation tables.

5.2 Design of Angluin's L* in the form of a flow chart

We have already seen the working of the L* algorithm proposed by Angluin. In this topic, we will see in detail how the design is carried out using a flowchart. The program is broken up into several parts and each part is explained in detail. The following is the flowchart for program.

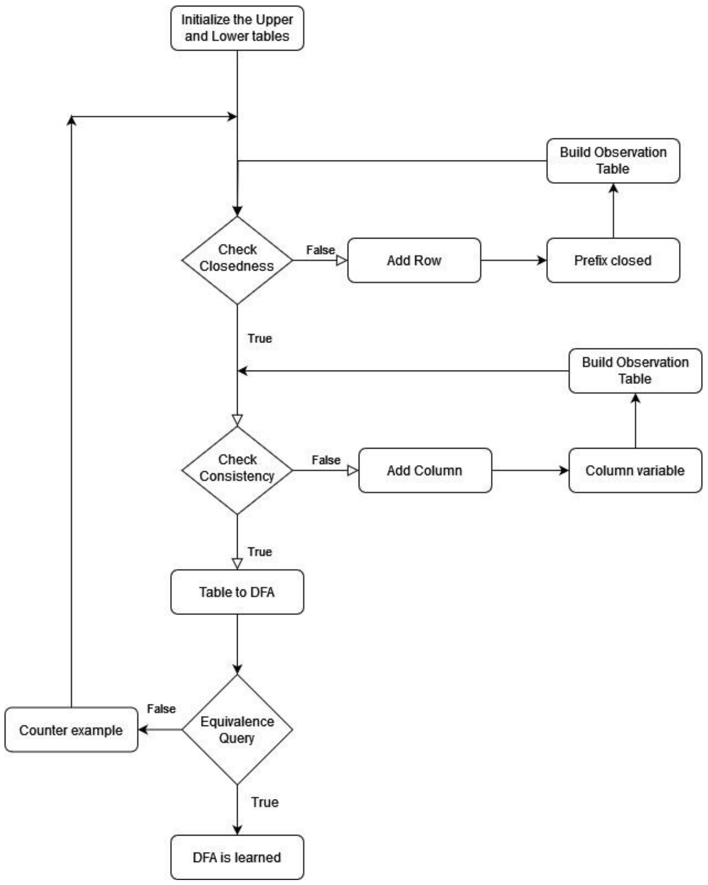

Figure 1: Flow chart for Angluin's L* algorithm

5.2.1 Check Closedness

This block checks the closedness of the Observation table. The function returns True or False depending on whether the table is closed or not. If the function returns True, we proceed forward with the next stage. Otherwise, we add another row, check for prefix-closedness and then continue checking the closedness for the newly created table.

5.2.2 Check Consistency

This block checks for the consistency of the observation table. The function returns True or False depending on whether the table is consistent or not. If the function returns True, we proceed onto the next stage, otherwise, we add another column. The function column variable will return a new column variable which satisfies the suffix closedness criteria. Then again, the table is checked for consistency and the process goes on until the table becomes consistent.

5.2.3 Table to DFA

The function Table to DFA, converts the given observation table into a DFA.

5.2.4 Equivalence Query

The DFA checks if it is equivalent to the actual DFA and if it returns True, the loop stops and returns the DFA learned. Otherwise, the Teacher will return a counter example and the process starts from the beginning.

5.3 Design of Rivest-Schapire's improvement on L*

5.3.1 Overview

It includes various helper functions and constants to interactively build the DFA by querying a membership oracle and an equivalence oracle. The learning process refines an observation table until the learned DFA matches the target DFA.

5.3.2 Dependencies

The program requires the following Python libraries:

- \`pandas\`
- \`numpy\`
- \`random\`
- \`string\`
- \`math\`

5.3.3 Constants

DFA Definitions	Multiple DFA definitions for different scenarios
ACTUAL TRANSITION	Transition function for the target DFA.
ACTUAL INITIAL	Initial state of the target DFA.
ACTUAL STATES	All states in the target DFA
ACTUAL FINAL	Final states in the target DFA.
EPSILON	Represents the empty string.

ALPHABET	List of symbols the DFA can process, including epsilon.
ACTUAL DFA	Tuple containing the DFA's transition function, initial state, final states, and all states.
COUNTER EXAMPLE	Stores counterexamples found during learning.
COUNT	Counts the number of membership queries made.

Table 2: List of constants for Rivest-Schapire's improvement on L*

5.3.4 Functions

Equivalence Query: It checks if the learned DFA is equivalent to the target DFA. We get the learned DFA as input. It tells whether a DFA is equivalent to the actual DFA.

Membership Query: It determines if a given input string is accepted by the DFA. It takes a string as an input. It will give ,1 if accepted, 0 otherwise.

Alphabet Check: It verifies that all characters in the input string belong to the specified alphabet. It take each alphabet in a string as the input. It gives output indicating if the string belongs to the alphabet.

Initialize: It creates and initializes the observation tables for the algorithm.

Build Observation Table: It fills the observation tables with results from membership queries.

Check Closedness: This function ensures all rows in one table are present in another table, verifying the table's closedness property.

Column Variable: This identifies the next column to add to the observation tables to maintain consistency.

Check Consistency: It ensures the observation table is consistent, meaning equivalent rows have equivalent transitions.

Add Row: It adds a new row to the observation table when it is not closed.

Add Column: It adds a new column to the observation table when it is not consistent.

Prefix Closed: It generates a set of strings that are prefix-closed.

Dataframe To Dfa: It converts the observation table into a DFA by mapping rows to states and transitions. It gives the learned DFA.

Output Sequence: The function returns the output sequence for a given state and sequence of inputs. It gives the output in 0s and 1s sequence.

Output State: It returns the output state of the DFA for a given initial state and sequence of inputs. It gives the output state as output.

Output Value: It returns the output value of the learned DFA from the initial state. It gives the output value (1 or 0).

Generate Counter Example: It generates strings for counterexamples. And stores them. It also checks if the generated counter example is not an already generated one and also, the output given by the actual and learned DFA are different.

Binary Search: It returns the two components of the string during the binary search. It gives the output, Low string, Character, High string.

Rivest: It performs the Rivest-Schapire improvement. It gives the updated observation tables.

Prefix Closed Rivest: It returns the prefix-closed set of strings.

Suffix Closed Rivest: Returns the suffix-closed set of strings.

5.3.5 Main Execution Flow

- Initialize the observation tables.
- Check consistency and closedness of the tables.
- Add columns and rows as needed to maintain consistency and closedness.
- Convert the observation table into a DFA.
- Perform equivalence query.
- If the learned DFA is not equivalent, refine the tables using Rivest-Schapire improvement.
- Repeat until the learned DFA matches the target DFA.
- Output the learned DFA and the count of membership queries.

5.4 Design of Rivest-Schapire’s improvement on L* as a flowchart

In this section, we will focus on the improvement proposed by Rivest-Schapire variant. The only place where the program differs from Angluin’s L* is in counterexample processing. In this case, we also make use of binary search to identify the right partition.

5.4.1 Rivest program block

The program block Rivest does the entire counter example processing. In this block, the new counter example is generated using random string generator and also the counter example is ensured to be in such a way that the output obtained from the learned DFA is different from the actual DFA.

5.4.2 Binary Search

Binary search is performed to identify that particular part of string where the output value differs for the actual and learned DFA. For this functions such as alpha, output sequence and output value are used. (They were not in-built functions and build from scratch).

5.4.3 Prefix and Suffix closedness and Build observation table

After the strings to be added are identified, the rows and columns are checked for prefix and suffix closedness respectively and the observation table is built.

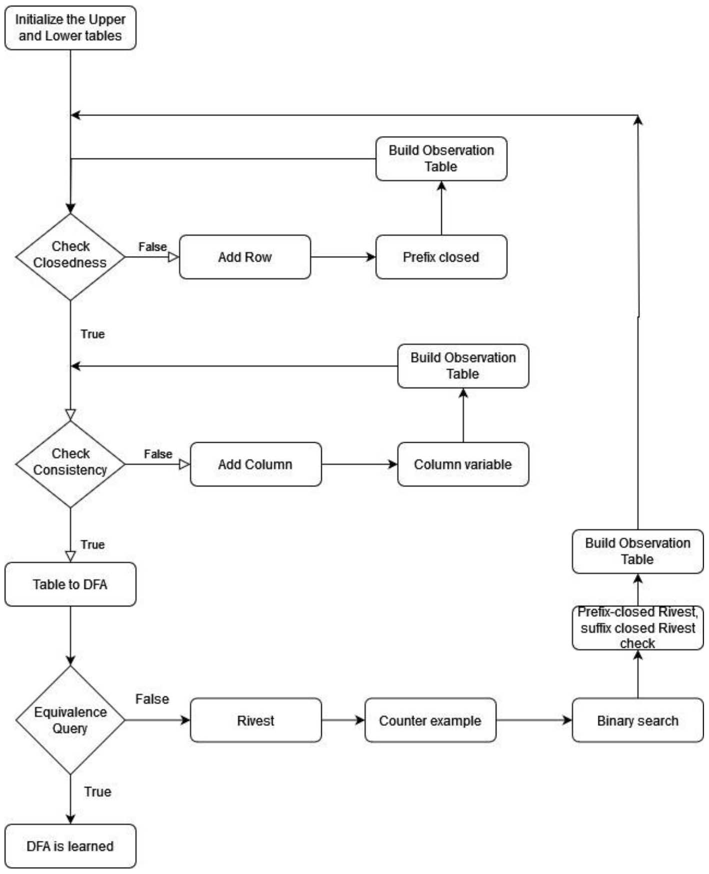

Figure 2: Flow chart for Rivest-Schapire variant of L*

5.5 Simulation of an environment

In this section we will discuss about the simulation of a simple DFA as a robot environment using Simulink software by MATLAB. We have replicated the DFA which accepts even number of a’s and b’s as a robot environment. The purpose of this environment is to tell us whether a robot is in a final state or not. It can be used in place of a membership query.

5.5.2 Robot sensors

In this simulation, the robot consists of four sensors. Sensor-1 which detects obstacles on the front, sensor-2 which detects obstacles on the back, sensor-3 which detects obstacles on the right and sensor-4 which detects obstacles in the left. And also, the robot consists of a compass to measure the degree of rotation. The following figure shows how the robot is modelled in the environment.

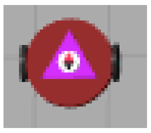

Figure 3 :Modelled Robot

5.5.3 Robot environment as a DFA

The DFA which accepts only even number of a’s and b’s is depicted as a robot environment below. Here, in this simulation, string ‘a’ denotes forward or reverse and the string ‘b’ denotes right or left. The DFA which accepts only even number of a’s and b’s is shown below.

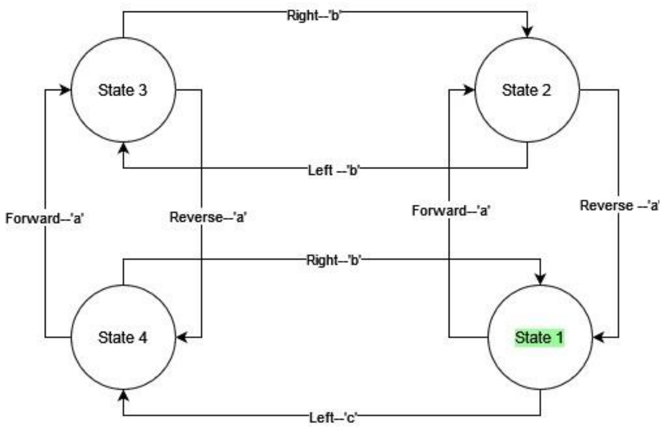

Figure 4: Robot environment as a DFA

Here, in the simulation there will be two buttons, forward/ reverse and right/left denoting ‘a’ and ‘b’ respectively. Assume that the robot is in the state 1, if we press the forward/ reverse button, the robot will move forward because it identifies that there is an obstacle on the back side and then moves straight. If it is state 2, it will identify using the back sensor that there is no obstacle at the back and move backward. The same applies for right and left sensor as well.

The final state is marked in green. The robot is modelled to be in one of the four states mentioned above. The robot will not be in any in-between states. The robot environment modelled in 3-D using Simulink is shown below.

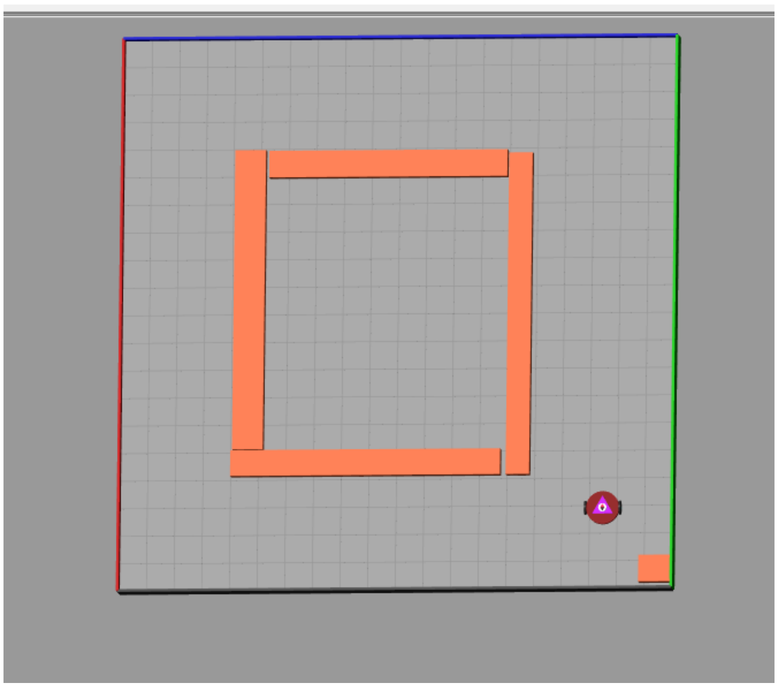

Figure 5: Robot environment using Simulink

Also, while interpreting the robot environment, it is advised to cross refer the DFA in Figure 4: Robot environment as a DFA. The current state of the robot is also interpreted using all the four sensors mentioned above. That is the front sensor and the left sensor should sense that there is no obstacle and also, the back and right sensor, should sense an obstacle.

In this simulation, the actual problems which could occur in a robot, that is for example, if a robot is asked to rotate to about 90 degrees, in real time environment, exact rotation to 90 degrees is impossible. There is a minor error. This simulation has taken all of those factors into account and mitigation methods were introduced to reduce those errors.

One more important factor to be taken into account is if we ask the robot to perform an action when another operation is already happening, the new action will not be performed. For e.g., when a robot is currently moving from state 1 to state 2, during that process, if a robot is mandated to move left or right, the robot will not perform the left or right. Only those actions, which were chosen at the time when it was at rest will be performed. This is taken care of the state flow chart in Simulink.

6 METHODOLOGY AND IMPLEMENTATION

6.1 Introduction

The implementation of L* proposed by Angluin and Rivest-Schapire is done in Python programming language. The python packages used were Numpy, Pandas, String, Math and Random. The code for the algorithms are below. It could be mentioned that the code is written from scratch. The Storage of a DFA as a python dictionary is taken as a reference from Stack overflow. [4] The design and explanation of the Simulink could be found in the previous chapter.

6.2 L* Algorithm

6.2.1 Pseudo code for Angluin's L* algorithm

The following is the pseudo code using which the implementation was done.

```
Start

    Initialize constants and actual DFA.
    Initialize COUNT to 0.

Initialize Observation Table

    Create Table1 and Table2 with initial indices and columns.
    Build Observation Table.

Main Loop:

    Check Closedness:

        If Table2 rows are not in Table1:

            Add missing rows from Table2 to Table1.
            Update Table2 indices.
            Rebuild Observation Table.

    Check Consistency:

        If the table is inconsistent:

            Add new columns to make the table consistent.
            Rebuild Observation Table.

    Construct DFA:

        Construct DFA from Observation Table.

    Equivalence Query:

        If the constructed DFA is equivalent to the actual DFA:

            Exit loop.

        Else:

            Add counterexample to the table.
            Update Observation Table.

End

    Print LearnedDFA, COUNT, Table1, and Table2.
```

6.2.2 Explanation of the pseudocode

Initialize constants and actual DFA:

This step sets up the initial conditions for the learning algorithm. The actual DFA (Deterministic Finite Automaton) is defined with its transition states, initial state, final states, and the alphabet.

Initialize COUNT to 0:

A counter variable COUNT is initialized to 0. This counter will be used to keep track of the number of membership queries made during the execution of the algorithm.

Initialize Observation Table:

This involves setting up two data tables, Table1 and Table2, that will be used to keep track of the observation sequences. These tables help in determining the states and transitions of the DFA. Table1 and Table2 are initialized with certain indices and columns based on the empty string (epsilon) and the alphabet characters.

Build Observation Table(Table1, Table2):

This step fills in the initial entries of Table1 and Table2 using membership queries. Membership queries determine whether a given string is accepted by the actual DFA.

Main Loop:

This starts an infinite loop that will continue until the learned DFA is equivalent to the actual DFA.

The Check Closedness function checks if Table2 rows (which are not yet included in Table1) have corresponding rows in Table1. If not, the table is not "closed." If the table is not closed, the missing rows from Table2 are added to Table1. The indices for Table2 are updated accordingly to ensure all new rows are accounted for. The observation table is rebuilt to reflect the new rows and ensure it remains accurate.

The Check Consistency function checks if the table is consistent. Consistency means that for any two rows in Table1 that are the same, the rows corresponding to any suffix appended to those rows are also the same. If the table is not consistent, new columns are added based on the inconsistencies found. The observation table is rebuilt again to include the new columns and maintain accuracy. Using the updated Table1 and Table2, a DFA is constructed. This involves determining the states, transitions, initial state, and final states of the DFA based on the observation table.

The Equivalence Query function checks if the constructed (learned) DFA is equivalent to the actual DFA. The learned DFA is equivalent to the actual DFA, the loop is exited, meaning the learning process is complete. If the learned DFA is not equivalent to the actual DFA. A counterexample (a string that is accepted by one DFA but not the other) is added to the table. The observation table is updated with the counterexample, which will help in refining the DFA in the next iterations.

End:

The learned DFA is printed, showing the final state of the learned automaton. The total number of membership queries made during the learning process is printed. The final state of Table1 is printed, showing the observation sequences and their membership status. The final state of

Table2 is printed, showing any remaining observation sequences that were not included in Table1.

6.2.3 Pseudo code for Rivest-Schapire variant of L*

```
Start

  Initialize constants and actual DFA.
  Initialize COUNT to 0.

Initialize Observation Table

  Create Table1 and Table2 with initial indices and columns.
  Build Observation Table.

Main Loop

  Check Consistency

    If Table1 and Table2 are not consistent:

      Add columns to make the table consistent.
      Rebuild Observation Table.

  Check Closedness

    If Table2 rows are not in Table1:

      Add missing rows from Table2 to Table1.
      Update Table2 indices.

  Rebuild Observation Table.

    Construct DFA
    Construct DFA from Observation Table.

  Equivalence Query

    If the learned DFA is equivalent to the actual DFA:

      Exit loop.
    Else:

      Add counterexample to the table.

  Update Observation Table.

    Rivest Schapire Improvement
    Get counterexample.
    Perform Binary Search on the counterexample.
    Add necessary rows and columns.
    Update Observation Table.

End

  Print LearnedDFA, COUNT, Table1, and Table2.
```

6.2.4 Explanation of pseudocode of Rivest-Schapire variant of L*

Initialization:

Initialize constants and actual DFA. Define the constants and the actual DFA that the learning algorithm will attempt to identify. Initialize a counter that will keep track of the number of queries or updates during the learning process. Initialize the Observation Table. Create Table1 and Table2. Table1 typically represents the observation table's rows with state information and inputs. Table2 holds the output of the queries, usually a function of the state and input pairs. Populate the observation table with initial data based on the states and inputs.

Check Consistency:

Consistency ensures that if two rows in Table1 are equivalent (i.e., they behave identically for all inputs), then they should also be equivalent in Table2. If the table is not consistent, add new columns to achieve consistency and rebuild the observation table.

Check Closedness:

Closedness ensures that all possible responses for all inputs are covered in Table1. If Table2 contains rows that are not in Table1, add these rows to Table1 and update the indices in Table2.

Construct DFA:

Once consistency and closedness are achieved, construct a DFA from the observation table. The states in the DFA are derived from the rows of Table1, and the transitions are derived from the observations.

Equivalence Query:

Check if the constructed DFA is equivalent to the actual DFA. If they are equivalent, the algorithm terminates. If not, use a counterexample to update the observation table by adding the counterexample to it and repeating the process.

Rivest Schapire Improvement:

Get counterexample:

When the learned DFA is not equivalent to the actual DFA, a counterexample is provided which shows where the DFA differs from the actual DFA.

Perform Binary Search on the counterexample:

Use binary search techniques to find the minimal counterexample, which can help in updating the observation table efficiently. Add necessary rows and columns. Based on the counterexample, add necessary rows and columns to the observation table to improve it. Adjust the observation table according to the new rows and columns derived from the counterexample.

End:

After the learning process is complete and the DFA is equivalent to the actual DFA, output the results including the learned DFA, the count of queries or updates, and the final states of Table1 and Table2.

6.3 Implementation of a DFA using Simulink

6.3.1 Simulation block diagram

The below is the Simulink program to create the robot environment model mentioned in the previous chapter. As mentioned, it consists of four sensors, one compass. It consists of two switches-Forward /Reverse and Left/ Right for ‘a’ and ‘b’ respectively. The Obstacle environment consists of the obstacles. The two motors are the feed for the two wheels. The forward and steer are for forward/ reverse motion and steering left or right respectively.

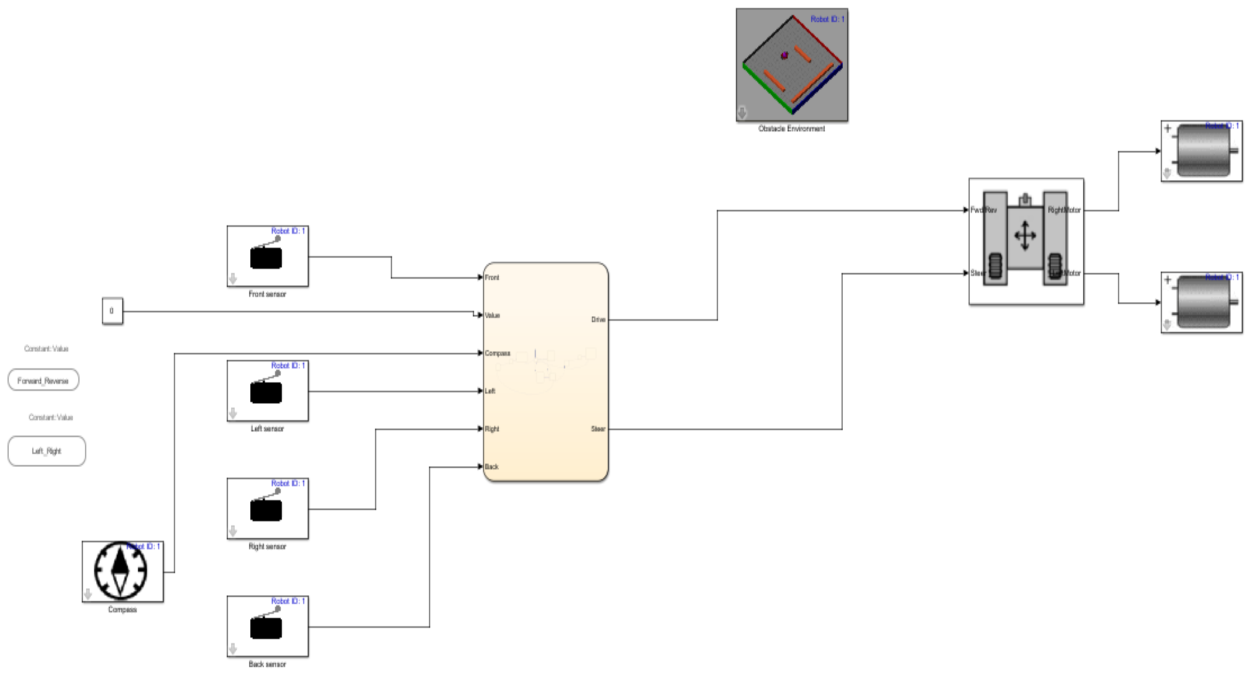

Figure 6: Simulation Overview in Simulink

The below is the state flow chart. This chart is necessary because, it takes care that another operation starts only after the completion of the current operation.

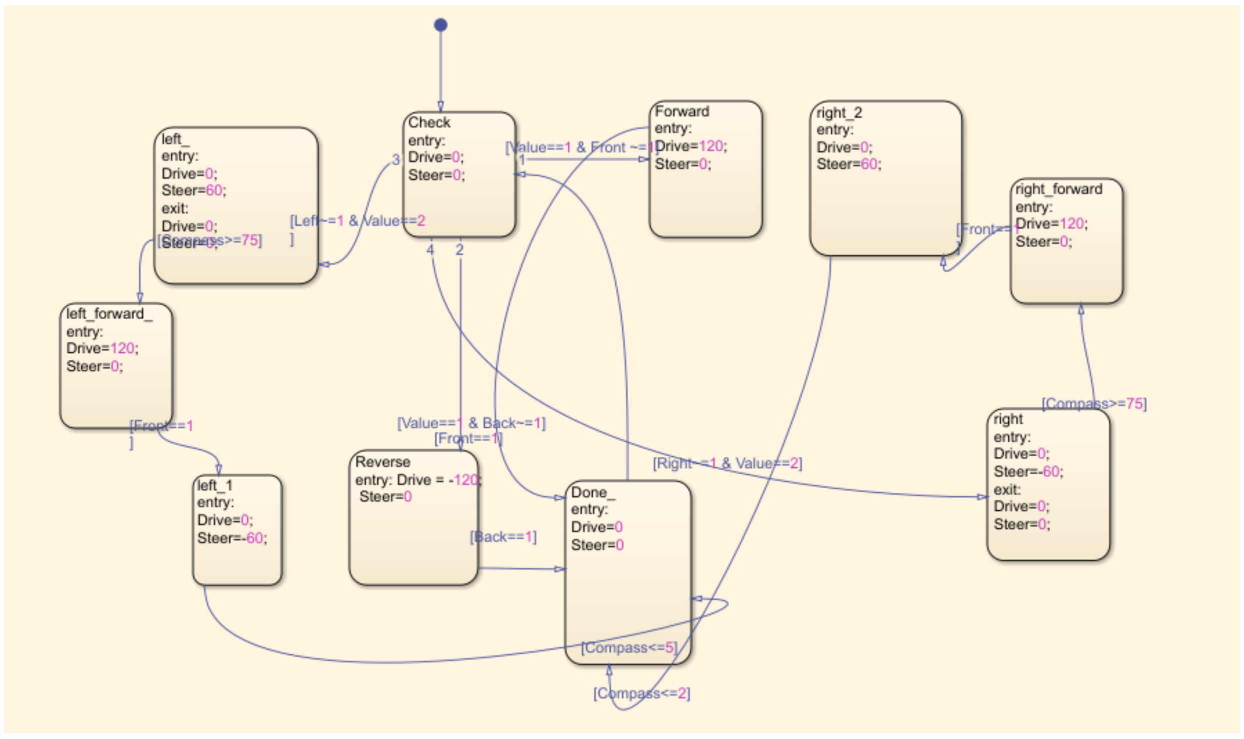

Figure 7: State flow chart

7 RESULTS AND ANALYSIS

7.1 Unit tests on L* by Angluin

The unit test conducted on each function of the program is listed below. Refer the section on Requirements and Specifications for the detailed working of each and every function listed below.

7.1.1 Equivalence Query Function

- Test when the learned DFA is equivalent to the actual DFA
- Test when the learned DFA is not equivalent to the actual DFA

7.1.2 Membership Query Function

- Test with strings that should be accepted by the DFA
- Test with strings that should be rejected by the DFA

7.1.3 Alphabet Check Function

- Test with strings that only contain characters from the alphabet
- Test with strings containing characters not in the alphabet

7.1.4 Initialize function

- Test the initial structure and content of the observation table

7.1.5 Build Observation Table Function

- Test if the observation table is correctly built with initial values
- Test the observation table with various input strings

7.1.6 Check Closedness Function

- Test with closed observation table.
- Test with non-closed observation table

7.1.7 Column Variable Function

- Test finding the variable to add as a new column when necessary
- Test returning `None` when no new column is needed

7.1.8 Check Consistency Function

- Test with consistent observation table
- Test with inconsistent observation table

7.1.9 Add Row Function

- Test adding a row to the observation table
- Test the observation table structure after adding a row

7.1.10 Add Column Function

- Test adding a column to the observation table
- Test the observation table structure after adding a column
- Test if the column added is the right one

7.1.11 Prefix Closed Function

- Test creating a prefix-closed set of strings from given row headers
- Test if the string generated is the right one in all cases

7.1.12 Dataframe To Dfa Function

- Test converting an observation table to a DFA:

7.1.13 Counter Example Function

- Test adding a counterexample to the observation table and updating it.
- Testing if a valid counterexample is generated and added.

7.2 Unit Tests on L* improvement by Rivest-Schapire

7.2.1 Equivalence Query function

- Test with a DFA that matches the actual DFA.
- Test with a DFA that does not match the actual DFA.

7.2.2 Membership Query Function

- Test with a string that is accepted by the DFA.
- Test with a string that is not accepted by the DFA.
- Test with an empty string.

7.2.3 Alphabet Check Function

- Test with a string that contains only characters in the alphabet.
- Test with a string that contains characters not in the alphabet.

7.2.4 Initialize Function

- Test if the tables are initialized correctly.

7.2.5 Build Observation Table Function

- Test if the tables are updated correctly with membership query results.

7.2.6 Check Closedness function

- Test with closed observation tables.
- Test with non-closed observation tables.

7.2.7 Column Variable function

- Test if the function returns the correct column variable to add.

7.2.8 Check Consistency function

- Test with consistent observation tables.
- Test with inconsistent observation tables.

7.2.9 Add Row function

- Test if the row is added correctly.

7.2.10 Add Column function

- Test if the column is added correctly.

7.2.11 Prefix Closed function

- Test if the prefix-closed set is correctly generated.

7.2.12 Data frame To Dfa function

- Test if the DFA is constructed correctly from the tables.

7.2.13 Output Sequence function

- Test if the output sequence is generated correctly.

7.2.14 Output State function

- Test if the final state is computed correctly.

7.2.15 Output Value function

- Test if the sequence is accepted correctly.

7.2.16 Generate Counter Example Function

- Test if new counterexamples are generated and added to the list.

7.2.17 Counter Example Function

- Test if a counterexample is generated correctly.

7.2.18 Alpha Function

- Test if the final state is correctly classified as accepting or not.

7.2.19 Binary Search Function

- Test if binary search finds distinguishing substrings correctly.

7.2.20 Rivest Function

- Test if the Rivest-Schapire algorithm updates the observation tables correctly.

7.2.21 Prefix closed Rivest Function

- Test if the prefix-closed set is generated correctly.

7.2.22 Suffix Closed Rivest Function

- Test if the suffix-closed set is generated correctly.

7.3 Evaluation of L* by Angluin and Rivest-Schapire

The program is tested using 6 different DFAs. The performance of L* is compared with the Rivest-Schapire variant on two criteria- the number of membership queries made and size of the observation table. The figure generated is from the following reference. [5]

7.4 Criteria for selecting the DFAs used for Testing

7.4.1 DFA that accepts any string that contains three consecutive ‘a’s

This DFA has no dead state. It is used as a sample to test if similar DFAs which does not have a dead state, works well. Also, this DFA was selected because it has a real-life applications such as recognizing specific behavioural patterns in robotic sensors or input sequences.

7.4.2 DFA that accepts even number of ‘a’s and ‘b’s

This DFA was mentioned as the example for explaining L* algorithm in Angluin’s paper. Also in robotics, tracking even numbers of certain actions or signals can be useful. Ensuring balanced and symmetrical movement patterns, such as an even number of forward and backward or left and right movements, can help maintain stability and prevent drift.

7.4.3 DFA that accepts any string

The DFA which accepts any string is chosen as an edge case to check how the algorithms behave for such DFAs. Also, in real robotic application, there may be modes where the robot should respond to any input without pre-defined restrictions.

7.4.4 DFA that accepts string which starts with ‘a’

It is a particular edge case which provides a simple boundary condition (whether the first character is 'a' or not), which is useful for testing how the algorithm handles such edge cases. It ensures prefix-handling that is the algorithm correctly processes prefixes of strings, which is crucial for the correct operation of more complex DFAs.

7.4.5 DFA that contains at least one dead state

Dead state inclusion ensures that the learning algorithm correctly handles dead states, which are states from which no accepting state can be reached. It also ensures input rejection, that is a DFA with a dead state explicitly tests the algorithm's ability to reject invalid strings, which is crucial for robust error handling. It ensures, graceful failure, that is, the system can handle and recover from incorrect inputs without crashing or producing invalid results.

7.4.6 DFA given in the paper published by Rivest-Schapire

This DFA was mentioned in the paper published by Rivest-Schapire to explain how homing sequences work and how L* is improved. Therefore, this was included in the test suite.

7.5 DFA test results

7.5.1 DFA that accepts any string that contains three consecutive 'a's

DFA:

The DFA that accepts any string that contains three consecutive 'a's is shown below. The below is the Observation table displayed as a pandas dataframe. The string 'e' denotes the 'epsilon' or the null value.

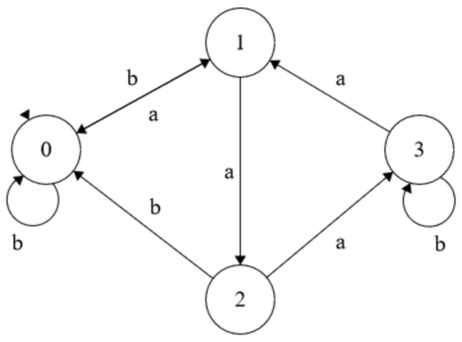

Figure 8: DFA that accepts string containing three consecutive 'a's.

Observation table obtained after Angluin's L* algorithm:

The below is the Observation table displayed as a pandas dataframe. The string 'e' denotes the 'epsilon' or the null value.

	e	a	aa
e	0	0	0
a	0	0	1
b	0	0	0
aa	0	1	0

ab	0	0	0
ba	0	0	1
bb	0	0	0
aaa	1	0	0
	e	a	aa
aab	0	0	0
aba	0	0	1
abb	0	0	0
baa	0	1	0
bab	0	0	0
bba	0	0	1
bbb	0	0	0
aaaa	0	0	1
aaab	1	0	0

Observation table obtained from Rivest-Schapire's variant of L*:

The below is the Observation table displayed as a pandas dataframe. The string 'e' denotes the 'epsilon' or the null value.

	e	aa	a
e	0	0	0
a	0	1	0
b	0	0	0
aa	0	0	1
ab	0	0	0
ba	0	1	0
bb	0	0	0
aaa	1	0	0
	e	aa	a
aab	0	0	0
aba	0	1	0
abb	0	0	0
baa	0	0	1
bab	0	0	0
bba	0	1	0
bbb	0	0	0
aaaa	0	1	0
aaab	1	0	0

Comparison of results:

The following is the table containing the result and comparison.

	Angluin's L* Algorithm	Rivest-Schapire Variant
Number of membership queries	526	505
Number of rows in the Observation Table	17	17
Number of columns in the Observation Table	3	3

Table 3: Comparison for the DFA that accepts the strings containing three consecutive 'a's

7.5.2 DFA that accepts any string that contains even number of 'a' and 'b'

DFA:

The DFA that accepts any string that contains even number of 'a' and 'b' is shown below.

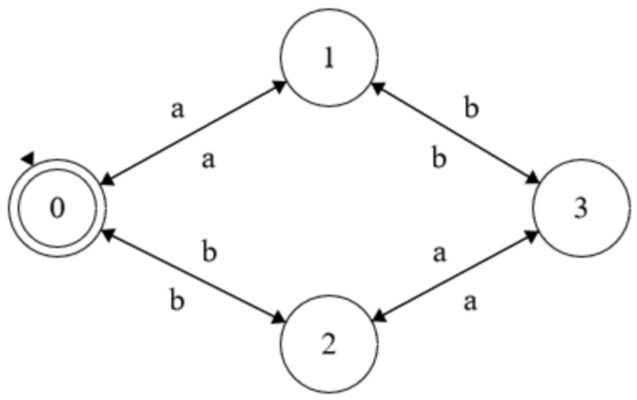

Figure 9:DFA that accepts string containing even number of 'a' and 'b'.

Observation table obtained after Angluin's L* algorithm:

The below is the Observation table displayed as a pandas dataframe. The string 'e' denotes the 'epsilon' or the null value.

```
              e      a        b
 e       1       0       0
```

```
a       0       1       0
b       0       0       1
aa      1       0       0
ab      0       0       0
ba      0       0       0
bb      1       0       0
        e       a       b
aaa     0       1       0
aab     0       0       1
aba     0       0       1
abb     0       1       0
baa     0       0       1
bab     0       1       0
bba     0       1       0
bbb     0       0       1
baa     0       0       1
bab     0       1       0
bba     0       1       0
bbb     0       0       1
bba     0       1       0
bbb     0       0       1
```

Observation table obtained from Rivest-Schapire's variant of L*:

The below is the Observation table displayed as a pandas data frame. The string 'e' denotes the 'epsilon' or the null value.

```
        e       a       b
e       1       0       0
a       0       1       0
b       0       0       1
ab      0       0       0
        e       a       b
aa      1       0       0
ba      0       0       0
bb      1       0       0
aba     0       0       1
abb     0       1       0
```

Comparison of results:

The following is the table containing the result and comparison.

	Angluin's L* Algorithm	**Rivest-Schapire Variant**
Number of membership queries	534	168
Number of rows in the Observation Table	15	9
Number of columns in the Observation Table	3	3

Table 4: Comparison for the DFA that accepts the strings containing even number of 'a' and 'b'

7.5.3 DFA that accepts any string

DFA:

The DFA that accepts any string is shown below.

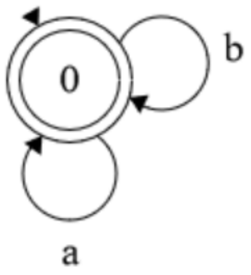

Figure 10: DFA that accepts any string.

Observation table obtained after Angluin's L* algorithm:

The below is the Observation table displayed as a pandas data frame. The string 'e' denotes the 'epsilon' or the null value.

```
          e
 e        1
          e
 a        1
```

```
b       1
```

Observation table obtained from Rivest-Schapire's variant of L*:

The below is the Observation table displayed as a pandas data frame. The string 'e' denotes the 'epsilon' or the null value.

```
        e
e       1
        e
a       1
b       1
```

Comparison of results:

The following is the table containing the result and comparison.

	Angluin's L* Algorithm	**Rivest-Schapire Variant**
Number of membership queries	9	9
Number of rows in the Observation Table	3	3
Number of columns in the Observation Table	1	1

Table 5: Comparison for the DFA that accepts any string.

7.5.4 DFA that accepts string which starts with 'a'

DFA:

The DFA that accepts strings that starts with 'a' is shown below.

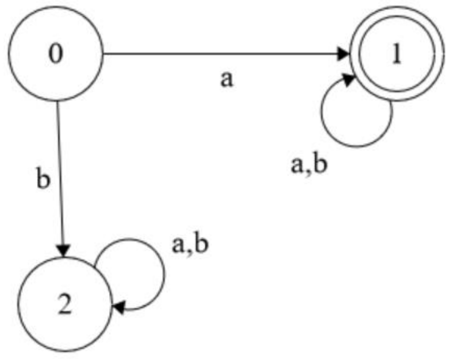

Figure 11: DFA that accepts strings that start with ‘a’

Observation table obtained after Angluin’s L* algorithm:

The below is the Observation table displayed as a pandas dataframe. The ‘e’ denotes the ‘epsilon’ or the null value.

	e	a
e	0	1
a	1	1
b	0	0
aa	1	1
	e	a
ab	1	1
ba	0	0
bb	0	0
aaa	1	1
aab	1	1
aaa	1	1
Aab	1	1

Observation table obtained from Rivest-Schapire’s variant of L*:

The below is the Observation table displayed as a pandas dataframe. The ‘e’ denotes the ‘epsilon’ or the null value.

	e	a
e	0	1
a	1	1
b	0	0

	e	a
aa	1	1
ab	1	1
ba	0	0
bb	0	0

Comparison of results:

The following is the table containing the result and comparison.

	Angluin's L* Algorithm	**Rivest-Schapire Variant**
Number of membership queries	121	75
Number of rows in the Observation Table	9	7
Number of columns in the Observation Table	2	2

Table 6: Comparison for the DFA that accepts strings starting with 'a'

7.5.5 DFA that contains at least one dead state

DFA:

The DFA that contains at least one dead state is shown below.

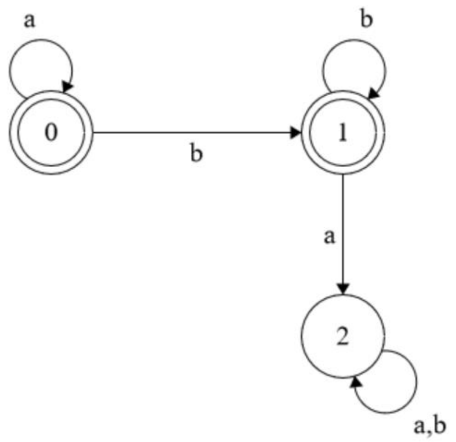

Figure 12: DFA that contains at least one dead state

Observation table obtained after Angluin's L* algorithm:

The below is the Observation table displayed as a pandas dataframe. The ‘e’ denotes the ‘epsilon’ or the null value.

```
        e      a
e       1      1
a       1      1
b       1      0
aa      1      1
ab      1      0
ba      0      0
        e      a
bb      1      0
aaa     1      1
aab     1      0
aba     0      0
abb     1      0
baa     0      0
bab     0      0
ba      0      0
```

Observation table obtained from Rivest-Schapire’s variant of L*:

The below is the Observation table displayed as a pandas dataframe. The ‘e’ denotes the ‘epsilon’ or the null value.

```
        e      a
e       1      1
b       1      0
a       1      1
ba      0      0
        e      a
bb      1      0
aa      1      1
ab      1      0
baa     0      0
bab     0      0
baa     0      0
bab     0      0
```

Comparison of results:

The following is the table containing the result and comparison.

	Angluin's L* Algorithm	**Rivest-Schapire Variant**
Number of membership queries	239	115
Number of rows in the Observation Table	13	9
Number of columns in the Observation Table	2	2

Table 7: Comparison for the DFA that contains at least one dead state

7.5.6 DFA given in the paper published by Rivest-Schapire [3]

DFA:

The DFA mentioned in the paper published by Rivest-Schapire [3] is given below.

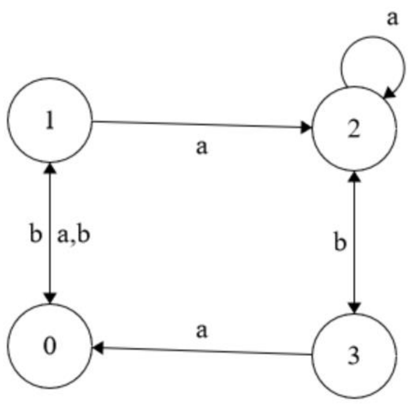

Figure 13: DFA mentioned in the paper published by Rivest-Schapire

Observation table obtained after Angluin's L* algorithm:

The below is the Observation table displayed as a pandas data frame. The 'e' denotes the 'epsilon' or the null value.

	e	a	ab
e	0	1	1
a	1	1	1
b	1	1	1

aa	1	1	0
ab	0	1	1
ba	1	1	0
bb	0	1	1
aaa	1	1	0
aab	0	0	1
aba	1	1	1
	e	a	ba
abb	1	1	1
baa	1	1	0
bab	0	0	1
bba	1	1	1
bbb	1	1	1
aaaa	1	1	0
aaab	0	0	1
aaba	0	1	1
aabb	1	1	0
abaa	1	1	0
abab	0	1	1
bba	1	1	1
bbb	1	1	1
aaaa	1	1	0
aaab	0	0	1
aaba	0	1	1
aabb	1	1	0
abaa	1	1	0
abab	0	1	1

Observation table obtained from Rivest-Schapire's variant of L*:

The below is the observation table displayed as a pandas data frame. The 'e' denotes the epsilon or the null value.

	e	ba	b	a
e	0	1	1	1
a	1	1	0	1
aa	1	0	0	1
aab	0	1	1	0
	e	ba	b	a

b	1	1	0	1
ab	0	1	1	1
aaa	1	0	0	1
aaba	0	1	1	1
aabb	1	0	0	1

Comparison of results:

The following is the table containing the result and comparison.

	Angluin's L* Algorithm	**Rivest-Schapire Variant**
Number of membership queries	934	234
Number of rows in the Observation Table	21	9
Number of columns in the Observation Table	3	4

Table 8: Comparison of results for the DFA mentioned in the paper published by Rivest-Schapire[3].

7.6 DFA Simulation

7.6.1 Result

As mentioned, the simulated Robot environment will give output, stating whether the given string belongs to a particular DFA or not. In other words, the result of the membership query. The result could either be that the robot is not in final state or it could be that the robot is in final state, based on the result of the simulated sensor on the robot. The following is the block diagram that gives the result of the membership query.

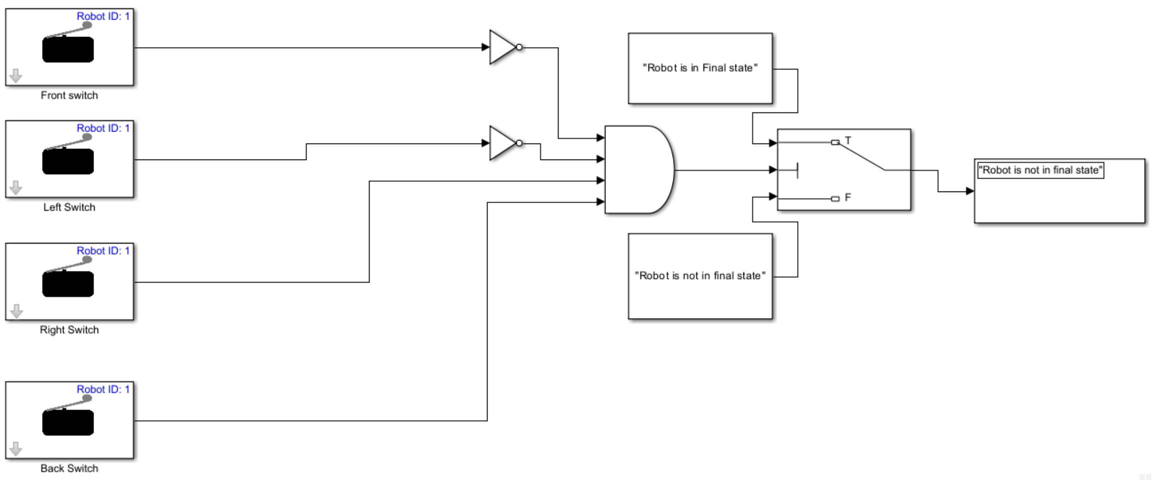

Figure 14: Membership Query using the simulation

8 EVALUATION

8.1 Significance of L* in the future

8.1.1 Efficiency and Practicality

L* operates in polynomial time with respect to the size of the minimal DFA and the length of the longest counterexample. This makes it efficient and practical for many applications, providing a balance between learning speed and accuracy. By constructing a minimal DFA, L* ensures that the resulting model is as simple as possible while accurately describing the target language. This simplicity is beneficial for analysis, verification, and implementation.

8.1.2 Wide Range of Applications

L* is used in formal verification to infer models of systems and check their correctness against specifications. It helps in discovering bugs and verifying properties of hardware and software systems. The algorithm has spurred significant research in automata theory, leading to the development of more advanced learning algorithms and variations that extend its applicability. In NLP, L* can be used to learn regular grammars for specific language tasks, aiding in tasks such as syntax analysis and pattern recognition.

8.1.3 Extensions and Improvements

L* has inspired numerous extensions and improvements, such as algorithms for learning non-deterministic finite automata (NFA), context-free grammars, and more complex structures. These advancements build on the foundational principles of L*. The principles of L* have been adapted for use in systems that require real-time learning and adaptation, such as robotics, where a system must quickly learn and adapt to changing environments.

8.1.4 Support for Structured Data Learning

L* is particularly well-suited for learning patterns in structured data, such as sequences and trees, making it applicable to a wide range of domains where such data is prevalent. The algorithm's approach to learning structured representations has influenced the development of more advanced techniques in machine learning and data mining.

8.2 L* as a tool for robot environment modelling

8.2.1 Learning and Adaptation

L* can be used to enable robots to learn about their environment and adapt to it. For example, a robot can learn the layout of a space and the rules of navigation within that space through interaction, allowing it to adapt to changes or new environments without human intervention. Also, by modelling the environment, a robot can learn the optimal paths and avoid obstacles more efficiently. This is crucial for tasks such as autonomous driving, warehouse automation, and service robots.

8.2.2 Verification and Validation

Using L* to model the robot's environment helps in verifying that the robot behaves as expected under different conditions. This is important for ensuring the safety and reliability of the robot, particularly in critical applications like medical surgery robots or autonomous vehicles. The L* algorithm can help identify and correct errors in the robot's behaviour by comparing the learned model against the expected model of the environment.

Also, by understanding the environment through a learned model, robots can optimize their task execution. For example, in manufacturing, robots can learn the most efficient ways to assemble parts or handle materials. Robots can also learn to manage resources like battery life or computational power by modelling and predicting the demands of their environment.

8.3 Significance of comparing the algorithms

8.3.1 Innovations and Extensions

The Rivest-Schapire improvement introduces new techniques or modifications to the original L* algorithm. Comparing them allows researchers and practitioners to understand these innovations and how they contribute to the overall performance. By identifying the strengths and weaknesses of both algorithms, researchers can develop further improvements or new algorithms that build on the best aspects of each.

8.3.2 Educational and Theoretical Insights

Comparing the two algorithms provides valuable insights into the design and development of learning algorithms. It showcases how theoretical advancements can lead to practical improvements. In an educational context, understanding the differences between the algorithms helps students grasp the evolution of learning techniques and appreciate the nuances of algorithm design. Practitioners can use the comparison to make informed decisions about which algorithm to use in specific applications, based on factors like efficiency, scalability, and robustness.

8.3.3 Use Case Evaluation

Different applications may benefit from different algorithms. For instance, some applications might prioritize speed over accuracy, while others might need highly accurate models even if they take longer to compute. Comparing L* and its improvement helps identify the best fit for various use cases. The comparison can reveal how each algorithm performs under specific conditions, such as varying data sizes, noise levels, or types of regular languages being learned.

9 LEGAL, SOCIAL, ETHICAL AND PROFESSIONAL ISSUES

9.1 Code of good practice

This project has been done completely done based on the principles highlighted in Code of Good practice issued by the British Computer Society (BSC). The research, design, implementation of the software, and all produced reports have duly reflected this rule. Various ideas from research publications in the field have been outlined in the report. In this context, careful attention has been given to sourcing and referencing each idea, piece of information prepared by third parties, ensuring that ownership of others' work is not claimed. It is also to be highlighted that the code was based on the algorithm proposed by Angluin and Rivest-Schapire. The issues with respect to a robot learning its environment in the form of a DFA is highlighted below.

9.2 Legal Issues

9.2.1 Data Privacy

Modeling a robot environment often involves collecting and processing large amounts of data. Ensuring compliance with data protection laws such as the General Data Protection Regulation (GDPR) in the EU or the California Consumer Privacy Act (CCPA) is essential. This includes obtaining consent from individuals whose data is collected, ensuring data is anonymized, and securing data against breaches.

9.2.2 Intellectual Property

Clarifying ownership rights over the data and the DFA model is critical. This involves understanding who owns the data collected by the robots and any intellectual property rights related to the DFA model. Adhering to licensing agreements for any third-party data or software used in the development process is also necessary.

9.2.3 Transparency and Accountability

Legal requirements often mandate transparency in how robotic systems operate. This includes documenting the development process, providing clear explanations of how decisions are made within the DFA model, and ensuring that there is accountability for any actions taken by the robot.

9.2.4 Safety and Liability

Ensuring that the robot operates safely within its environment is a legal necessity. This involves rigorous testing and adherence to safety standards to prevent harm. Additionally, establishing liability for any damages or injuries caused by the robot is essential, which can involve complex legal considerations.

9.3 Ethical Issues

9.3.1 Transparency and Explainability

The DFA model should be transparent and explainable. Users and stakeholders should understand how the model works and the basis for its decisions. This transparency helps build trust and ensures that the model can be scrutinized and improved over time.

9.3.2 Accountability and Redress

Establishing clear lines of accountability for the actions and decisions made by the robot is important. There should be mechanisms in place for individuals to challenge and seek redress for any adverse outcomes resulting from the robot’s actions.

9.3.3 Security

Ensuring robust security measures to protect the data and the integrity of the DFA model is essential. This includes safeguarding against cyber-attacks, unauthorized access, and ensuring that the robot’s operations cannot be maliciously manipulated.

9.3.4 Impact on Employment

The deployment of robots modelled using DFA may impact employment, particularly in sectors where robots might replace human workers. Ethical considerations include addressing job displacement and investing in retraining and skill development for affected workers.

9.3.5 Social Impact

Evaluating the broader social impact of deploying robots is necessary. This includes ensuring that the technology benefits society as a whole and does not disproportionately harm marginalized groups. Ethical use of the technology should aim to enhance social good and avoid exacerbating existing inequalities.

9.4 Mitigation Strategies

To address these legal and ethical issues, several mitigation strategies can be employed. Regular audits can help detect and correct biases in the DFA model. Developing and adhering to a robust set of ethical guidelines ensures responsible development and deployment. Engaging with diverse stakeholders throughout the development process helps identify potential issues and address them proactively. Continuous monitoring and evaluation of the robot's performance in real-world settings ensure it operates as intended. Transparency reports detailing the robot's performance, decision-making processes, and any incidents help maintain accountability and trust. By addressing these legal and ethical issues, organizations can responsibly develop and deploy robotic systems modelled using DFA.

10 CONCLUSION

10.1 Results

From the results, it is evident that the number of rows in the Observation table and the number of membership queries have significantly reduced in most cases, highlighting the efficiency improvements achieved through the implemented modifications. However, in certain examples, such as the DFA that accepts all possible strings, there is no significant variation between the original and improved algorithms. This indicates that for specific types of DFAs, the optimizations may not provide substantial benefits.

10.2 Future Improvements

10.2.1 L* Algorithm by Angluin and Rivest-Schapire

While substantial progress has been made, it is well-known that it is not possible to identify all potential bugs in any software application. Consequently, the absence of defects in the code cannot be guaranteed. Additionally, the current implementation may exhibit higher-order exponential runtime, which poses challenges for scalability. Future work could focus on the following areas:

Comprehensive Testing:

We can focus on expanding the testing to include a wider variety of DFAs. By testing with diverse and complex DFAs, we can better understand the algorithm’s strengths and weaknesses. Ensure that edge cases and unusual patterns are thoroughly tested to verify the robustness of the algorithm.

Scalability Improvements:

We can focus on algorithm optimization. Explore optimization techniques to improve the scalability of the algorithm. Efficient data structures could be used. Utilizing more sophisticated data structures to manage states and transitions more efficiently. Implementing parallel processing techniques to handle large state spaces more effectively. Incorporating heuristics and approximation methods to reduce computational overhead without significantly compromising accuracy. Advanced techniques could be used to investigate advanced algorithmic improvements and heuristics that can reduce runtime and resource consumption.

We can also develop mechanisms for dynamically adjusting the learning process based on real-time performance metrics and feedback.

Robustness and Reliability:

We can also focus on making the implementation more robust by incorporating comprehensive error handling and debugging mechanisms. We can also ensure the reliability of the learning algorithm through extensive testing and validation, including stress tests and real-world scenario simulations.

10.2.2 Robot Environment Simulation

In this study, a robot environment resembling a DFA that accepts an even number of 'a's and 'b's was simulated. The purpose of this environment was to determine whether the current state is an accepting state, effectively serving as a membership query. The future improvements could include the following.

MATLAB Implementation:

We could implement the entire algorithm in MATLAB, leveraging its robust simulation capabilities and extensive libraries. This would provide a more integrated environment for simulating and testing the robot environment. Also, we could use advanced Simulink Models and utilize advanced features of Simulink to create more realistic and complex models of the robot environment.

Enhanced Robot Simulation:

Integrated teacher role could have been used. We could use the simulated robot environment in place of a teacher for the membership query, aligning with the approach mentioned by Rivest and Schapire in their paper. This integration can improve the efficiency and accuracy of the learning process. Also, complex behaviours such as develop more sophisticated robot behaviours and interactions to represent complex state transitions.

Complex DFA Simulations:

We could simulate more complex DFAs with a greater number of states than the current implementation. This would involve designing intricate robot behaviours and sensor interactions to represent complex state transitions.

Real-World Application:

We could incorporate practical scenarios. We could explore the applicability of the robot environment simulation in real-world scenarios, such as automated navigation, obstacle avoidance, and task execution. Also incorporate adaptive learning and implement machine learning techniques to improve the robot's adaptability and learning capabilities in dynamic environments.

Interoperability:

Develop interfaces that allow for seamless interoperability between several versions implementations, enabling researchers to leverage the strengths of both platforms. We can also ensure that the simulated environment can be easily integrated with other software tools and frameworks used in robotics and automata learning research.

User Interface and Visualization:

Enhance the user interface to provide visual feedback on the state transitions and DFA learning process, making it easier to understand and debug. Also, develop interactive simulation tools that allow users to manually input strings and observe the robot's behaviour in real-time.

Performance Metrics:

We could establish comprehensive performance metrics and benchmarks to evaluate the efficiency and accuracy of the learning algorithms and simulations. And implement a framework for continuous improvement based on performance data and user feedback.

By addressing these areas, future work can significantly enhance the performance, scalability, and practical applicability of the L* algorithm and the robot environment simulation, paving the way for more advanced and robust implementations.

11 REFERENCES

[1] Rick ten Tije, "A comparison of counterexample processing techniques in Angluin-style learning algorithms"

[2] Dana Angluin ,"Learning regular sets from queries and Counterexamples"

[3] Ronald. L. Rivest and Robert. E. Schapire ,"Inference of Finite Automata using Homing sequences"

[4] Stack Overflow website, "How to implement a DFA in python", https://stackoverflow.com/questions/35272592/how-are-finite-automata-implemented-in-code

[5] Finite State Machine Designer, "Graphic representation of a DFA", https://madebyevan.com/fsm/

[6] Mathworks,"Virtual-Robot-Environment", https://www.mathworks.com/videos/matlab-and-simulink-pass-competitions-hub-getting-started-with-robotics-playground-virtual-worlds-1533569380647.html

[7] Rivest and Schapire ,"A new approach to unsupervised learning", 1987

[8] Pitt and Warmuth ,"The minimum consistent DFA problem cannot be approximated within any polynomial", 1993

[9] Zvi Kohavi ,"Switching and Finite Automata Theory", 1978

[10] Rivest and Schapire , "Diversity Based representation", 1988

12 APPENDICES

12.1 Appendix A: Source Code for L*

12.1.1 Structure Overview

The program file consists of two python files, Angluin's L* and Rivest-Schapire variant of L*. The python files Angluin.py and RivestSchapire.py were the files which were used for implementation.

The code consists of six different DFAs to test out the working. They were commented out and we can un-comment each and every one to test out different types of it. It is preferred to un comment any one DFA at a particular. Since constants can never be defined in Python, the values of those variables which are not supposed to change during execution are denoted in upper case letters. Also, the running of the code involves installation of packages such as Pandas, Numpy, String, Math and Random.

12.1.2 Python code for Angluin's L*

```
# Constants

# 1 # DFA to accept three consecutive 'a's

# ACTUALTRANSITION={0:{'':0,'a':1, 'b':0},
#      1:{'':1,'a':2, 'b':0},
#      2:{'':2,'a':3, 'b':0},
#      3:{'':3,'a':1,'b':3}}
# ACTUALINITIAL=0
# ACTUALSTATES=[0,1,2,3]
# ACTUALFINAL=[3]

# 2 # DFA to accept even number of 'a's and 'b's

ACTUALTRANSITION={0:{'':0,'a':1, 'b':2},
     1:{'':1,'a':0, 'b':3},
     2:{'':2,'a':3, 'b':0},
     3:{'':3,'a':2,'b':1}
}
ACTUALINITIAL=0
ACTUALSTATES=[0,1,2,3]
ACTUALFINAL=[0]

# 3 # DFA that accepts any string to check edge case

# ACTUALTRANSITION={0:{'':0,'a':0, 'b':0}
# }
# ACTUALINITIAL=0
# ACTUALSTATES=[0]
# ACTUALFINAL=[0]

# 4 DFA that starts with 'a'

# ACTUALTRANSITION={0:{'':0,'a':1, 'b':2},
#      1:{'':1,'a':1, 'b':1},
#      2:{'':2,'a':2, 'b':2}
# }
# ACTUALINITIAL=0
# ACTUALSTATES=[0,1,2]
# ACTUALFINAL=[1]

# 5 DFA that accepts only the input a few inputs with dead states

# ACTUALTRANSITION={0:{'':0,'a':0, 'b':1},
#      1:{'':1,'a':2, 'b':1},
#      2:{'':2,'a':2, 'b':2}
# }
# ACTUALINITIAL=0
# ACTUALSTATES=[0,1,2]
# ACTUALFINAL=[0,1]

# DFA mentioned in the paper published by Rivest Schapire

# ACTUALTRANSITION={0:{'':0,'a':1, 'b':1},
#      1:{'':1,'a':2, 'b':0},
#      2:{'':2,'a':2, 'b':3},
#      3:{'':3,'a':0,'b':2}}
# ACTUALINITIAL=0
# ACTUALSTATES=[0,1,2,3]
# ACTUALFINAL=[1,2]

# Constants

EPSILON=''
ALPHABET=[EPSILON,'a','b']
ACTUALDFA=[ACTUALTRANSITION,ACTUALINITIAL,ACTUALFINAL,ACTUALSTATES]
COUNTEREXAMPLE=['']
COUNT=0


# Equivalence query: To check whether two DFAs are equal

def EquivalenceQuery(Learned):

    if(Learned==ACTUALDFA):
        return True
    else:
        return False

# Membership query: To check whether an input string is a part of the DFA

def MembershipQuery(String):
    global COUNT
    COUNT+=1

    if (AlphabetCheck(String,ALPHABET)==True):

        CurrState = ACTUALDFA[1]
        FinalState=ACTUALDFA[2]
        dfa=ACTUALDFA[0]

        for Char in String:
            CurrState = dfa[CurrState][Char]

        return int(CurrState in FinalState)

    else:
        return 0
```

```
# Alphabet check : To check whether an input string is a part of a particular
alphabet

def AlphabetCheck(String,ALPHABET):
    Alpha=True

    for Char in String:
        if Char in ALPHABET:
            Alpha=True
            continue
        else:
            Alpha =False
            break

    return Alpha

# Initialise the Observation table

def Initialize():
    Columns=[EPSILON]
    Index1=[EPSILON]
    Index2=['a','b']

    Table1 = pd.DataFrame(columns=Columns,index=Index1)
    Table2 = pd.DataFrame(columns=Columns,index=Index2)

    return BuildObservationTable(Table1,Table2)

# Filling up the entries of Observation table using membership queries

def BuildObservationTable(Table1,Table2):
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Columns=list(Table1.columns)

    Array1=np.array(Table1)
    Array2=np.array(Table2)

    for i in range (len(Index1)):
        for j in range (len(Columns)):
            Array1[i][j]=MembershipQuery(Index1[i]+Columns[j])
            Array1[i][j]=MembershipQuery(Index1[i]+Columns[j])

    for i in range (len(Index2)):
        for j in range (len(Columns)):
            Array2[i][j]=MembershipQuery(Index2[i]+Columns[j])

            Array2[i][j]=MembershipQuery(Index2[i]+Columns[j])

    Table1_new = pd.DataFrame(Array1,columns=Columns,index=Index1)
    Table2_new = pd.DataFrame(Array2,columns=Columns,index=Index2)

    return Table1_new,Table2_new

# Check the closedness of the observation Table

def CheckClosedness(Table1,Table2):
    Array1=np.array(Table1)
    Array2=np.array(Table2)

    for i in range(len(Array2)):
        if Array2[i] not in Array1:
            return False

    return True

# Variable to be chosen for the next column

def ColumnVariable(Table1,Table2):
    Array1=np.array(Table1).tolist()
    Array2=np.array(Table2).tolist()
    Array=Array1+Array2
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Index=Index1+Index2
    Columns=list(Table1.columns)

    compare=[]

    for i in range (len(Array1)-1):
        j=i+1
        while(j<len(Array1)):
            if Array1[i]==Array1[j]:
                compare.append([i,j])
            j=j+1

    for i in range (len(compare)):
        for j in range (len(ALPHABET)):
            for k in range (len(Columns)):
                value1=MembershipQuery(Index1[compare[i][0]]+ALPHABET[j]+Colum
ns[k])
                value2=MembershipQuery(Index1[compare[i][1]]+ALPHABET[j]+Colum
ns[k])

                if value1!=value2:
                    value=ALPHABET[j]+Columns[k]
                    if value not in Columns:
                        return value

    return None

# Check the consistency of the table

def CheckConsistency(Table1,Table2):
    Array1=np.array(Table1).tolist()
    Array2=np.array(Table2).tolist()
    Array=Array1+Array2
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Index=Index1+Index2
    Columns=list(Table1.columns)

    compare=[]

    for i in range (len(Array1)-1):
        j=i+1
        while(j<len(Array1)):
            if Array1[i]==Array1[j]:
                compare.append([i,j])
            j=j+1

    for i in range (len(compare)):
        for j in range (len(ALPHABET)):
            for k in range (len(Columns)):
                value1=MembershipQuery(Index1[compare[i][0]]+ALPHABET[j]+Colum
ns[k])
                value2=MembershipQuery(Index1[compare[i][1]]+ALPHABET[j]+Colum
ns[k])

                if value1!=value2:
                    return False
    return True

# Add a row in the upper side of the table

def AddRow(Table1,Table2):

    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Columns=list(Table1.columns)

    Array1=np.array(Table1).tolist()
    Array2=np.array(Table2).tolist()

    Array1.append(Array2[0])
    Array2.pop(0)

    Index1.append(Index2[0])
    Index2=PrefixClosed(Index1)

    Table1_new = pd.DataFrame(columns=Columns,index=Index1)
    Table2_new = pd.DataFrame(columns=Columns,index=Index2)

    T1,T2=BuildObservationTable(Table1_new,Table2_new)

    return T1,T2

# Add a column in the table

def AddColumn(Table1,Table2):
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Columns=list(Table1.columns)

    Append=ColumnVariable(Table1,Table2)
    if Append!=None:
        Columns.append(Append)

        Table1_new = pd.DataFrame(columns=Columns,index=Index1)
        Table2_new = pd.DataFrame(columns=Columns,index=Index2)
        T1,T2=BuildObservationTable(Table1_new,Table2_new)

        return T1,T2

    else:
        return Table1,Table2

# Create a prefix closed set of strings from the given row headers

def PrefixClosed(label):
    Final=[]

    for i in range (len(label)):
        for j in range (len(ALPHABET)):
            temp=label[i]+ALPHABET[j]
            if (temp not in Final)&(temp not in label):
                Final.append(label[i]+ALPHABET[j])

    return Final

# Construct DFA from Dataframe

def DataframetToDfa(Table1,Table2):
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Index=Index1+Index2
    Columns=list(Table1.columns)

    Array1=np.array(Table1).tolist()
    Array2=np.array(Table2).tolist()
    Array=Array1+Array2

    LearnedTransition_test={}
    LearnedInitial_test=0
    LearnedFinal_test=[]
    LearnedStates_test=[]
    LearnedRows_test=[]
    LearnedStrings_test=[]
    row=0
    for i in range (len(Index1)):
        if Array1[i] not in LearnedRows_test:
            LearnedStates_test.append(row)
            row=row+1
            LearnedRows_test.append(Array1[i])
            LearnedStrings_test.append(Index1[i])

    for i in range (len(LearnedStates_test)):
        temp={}
        for j in range (len(ALPHABET)):
            String=LearnedStrings_test[i]+ALPHABET[j]
            I=Index.index(String)
            Row=Array[I]
            Append=LearnedRows_test.index(Row)

            temp[ALPHABET[j]]=Append

        LearnedTransition_test[i]=temp

    for i in range (len(LearnedStates_test)):
        for j in range (len(ALPHABET)):
            String=LearnedStrings_test[i]+ALPHABET[j]
            I=Index.index(String)
            Row=Array[I]
            State=LearnedRows_test.index(Row)
            Append=LearnedStrings_test[State]
            if (MembershipQuery(Append)==1)&(State not in 
LearnedFinal_test)&(State in LearnedStates_test):
                LearnedFinal_test.append(State)


    LearnedDfa_test=[LearnedTransition_test,LearnedInitial_test,LearnedFinal_t
est,LearnedStates_test]

    return (LearnedDfa_test)

# Add Counter example and retrun the tables

def CounterExample(Table1,Table2):
    T1,T2=Table1,Table2

    while(1):
        T1,T2=AddRow(T1,T2)
        List=list(T1.index)

        if MembershipQuery(List[len(List)-1])==1:
            break

    return T1,T2

# Main

Table1,Table2=Initialize()
LearnedDfa=[]

while(1):

    while(CheckClosedness(Table1,Table2)!=True):
        Table1,Table2=AddRow(Table1,Table2)

    while(CheckConsistency(Table1,Table2)!=True):
        Table1,Table2=AddColumn(Table1,Table2)

    LearnedDfa=DataframetToDfa(Table1,Table2)

    if (EquivalenceQuery(LearnedDfa))!=True:
        Table1,Table2=CounterExample(Table1,Table2)
    else:
        break

print(LearnedDfa)
print(COUNT)
print(Table1)
print(Table2)
```

12.1.3 Rivest-Schapire's L* algorithm

```
# Constants

# 1 # DFA to accept three consecutive 'a's
# ACTUALTRANSITION={0:{'':0,'a':1, 'b':0},
#       1:{'':1,'a':2, 'b':0},
#       2:{'':2,'a':3, 'b':0},
#       3:{'':3,'a':1,'b':3}}
# ACTUALINITIAL=0
# ACTUALSTATES=[0,1,2,3]
# ACTUALFINAL=[3]

# 2 # DFA to accept even number of 'a's and 'b's

ACTUALTRANSITION={0:{'':0,'a':1, 'b':2},
     1:{'':1,'a':0, 'b':3},
     2:{'':2,'a':3, 'b':0},
     3:{'':3,'a':2,'b':1}
}
ACTUALINITIAL=0
ACTUALSTATES=[0,1,2,3]
ACTUALFINAL=[0]

# 3 # DFA that accepts any string to check edge case

# ACTUALTRANSITION={0:{'':0,'a':0, 'b':0}
# }
# ACTUALINITIAL=0
# ACTUALSTATES=[0]
# ACTUALFINAL=[0]

# 4 DFA that starts with 'a'

# ACTUALTRANSITION={0:{'':0,'a':1, 'b':2},
#       1:{'':1,'a':1, 'b':1},
#       2:{'':2,'a':2, 'b':2}
# }
# ACTUALINITIAL=0
# ACTUALSTATES=[0,1,2]
# ACTUALFINAL=[1]

# 5 DFA that accepts only the input a few inputs with dead states

# ACTUALTRANSITION={0:{'':0,'a':0, 'b':1},
#       1:{'':1,'a':2, 'b':1},
#       2:{'':2,'a':2, 'b':2}
# }
# ACTUALINITIAL=0
# ACTUALSTATES=[0,1,2]
# ACTUALFINAL=[0,1]

# DFA mentioned in the paper published by Rivest Schapire

# ACTUALTRANSITION={0:{'':0,'a':1, 'b':1},
#       1:{'':1,'a':2, 'b':0},
#       2:{'':2,'a':2, 'b':3},
#       3:{'':3,'a':0,'b':2}}
# ACTUALINITIAL=0
# ACTUALSTATES=[0,1,2,3]
# ACTUALFINAL=[1,2]

# Constants

EPSILON=''
ALPHABET=[EPSILON,'a','b']
ACTUALDFA=[ACTUALTRANSITION,ACTUALINITIAL,ACTUALFINAL,ACTUALSTATES]
COUNTEREXAMPLE=['']
COUNT=0


# Equivalence query: To check whether two DFAs are equal

def EquivalenceQuery(Learned):

    if(Learned==ACTUALDFA):
        return True
    else:
        return False

# Membership query: To check whether an input string is a part of the DFA

def MembershipQuery(String):
    global COUNT
    COUNT+=1

    if (AlphabetCheck(String,ALPHABET)==True):
        CurrState = ACTUALDFA[1]
        FinalState=ACTUALDFA[2]
        dfa=ACTUALDFA[0]

        for Char in String:
            CurrState = dfa[CurrState][Char]

        return int(CurrState in FinalState)

    else:
        return 0

# Alphabet check : To check whether an input string is a part of a particular
alphabet

def AlphabetCheck(String,ALPHABET):
    Alpha=True

    for Char in String:
        if Char in ALPHABET:
            Alpha=True
            continue
        else:
            Alpha =False
            break

    return Alpha

# Initialise the Observation table

def Initialize():
    Columns=[EPSILON]
    Index1=[EPSILON]
    Index2=['a','b']

    Table1 = pd.DataFrame(columns=Columns,index=Index1)
    Table2 = pd.DataFrame(columns=Columns,index=Index2)

    return BuildObservationTable(Table1,Table2)

# Filling up the entries of Observation table using membership queries
```

```python
def BuildObservationTable(Table1,Table2):
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Columns=list(Table1.columns)

    Array1=np.array(Table1)
    Array2=np.array(Table2)

    for i in range (len(Index1)):
        for j in range (len(Columns)):
            Array1[i][j]=MembershipQuery(Index1[i]+Columns[j])

            Array1[i][j]=MembershipQuery(Index1[i]+Columns[j])

    for i in range (len(Index2)):
        for j in range (len(Columns)):
            Array2[i][j]=MembershipQuery(Index2[i]+Columns[j])

            Array2[i][j]=MembershipQuery(Index2[i]+Columns[j])

    Table1_new = pd.DataFrame(Array1,columns=Columns,index=Index1)
    Table2_new = pd.DataFrame(Array2,columns=Columns,index=Index2)

    return Table1_new,Table2_new

# Check the closedness of the observation Table

def CheckClosedness(Table1,Table2):
    Array1=np.array(Table1)
    Array2=np.array(Table2)

    for i in range(len(Array2)):
        if Array2[i] not in Array1:
            return False

    return True

# Variable to be chosen for the next column

def ColumnVariable(Table1,Table2):
    Array1=np.array(Table1).tolist()
    Array2=np.array(Table2).tolist()
    Array=Array1+Array2
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Index=Index1+Index2
    Columns=list(Table1.columns)

    compare=[]

    for i in range (len(Array1)-1):
        j=i+1
        while(j<len(Array1)):
            if Array1[i]==Array1[j]:
                compare.append([i,j])
            j=j+1

    for i in range (len(compare)):
        for j in range (len(ALPHABET)):
            for k in range (len(Columns)):
                value1=MembershipQuery(Index1[compare[i][0]]+ALPHABET[j]+Colum
ns[k])
                value2=MembershipQuery(Index1[compare[i][1]]+ALPHABET[j]+Colum
ns[k])

                if value1!=value2:
                    value=ALPHABET[j]+Columns[k]
                    if value not in Columns:
                        return value

    return None

# Check the consistency of the table

def CheckConsistency(Table1,Table2):
    Array1=np.array(Table1).tolist()
    Array2=np.array(Table2).tolist()
    Array=Array1+Array2
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Index=Index1+Index2
    Columns=list(Table1.columns)

    compare=[]
    for i in range (len(Array1)-1):
        j=i+1
        while(j<len(Array1)):
            if Array1[i]==Array1[j]:
                compare.append([i,j])
            j=j+1

    for i in range (len(compare)):
        for j in range (len(ALPHABET)):
            for k in range (len(Columns)):
                value1=MembershipQuery(Index1[compare[i][0]]+ALPHABET[j]+Colum
ns[k])
                value2=MembershipQuery(Index1[compare[i][1]]+ALPHABET[j]+Colum
ns[k])

                if value1!=value2:
                    return False
    return True

# Add a row in the upper side of the table

def AddRow(Table1,Table2):

    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Columns=list(Table1.columns)

    Array1=np.array(Table1).tolist()
    Array2=np.array(Table2).tolist()

    Array1.append(Array2[0])
    Array2.pop(0)

    Index1.append(Index2[0])
    Index2=PrefixClosed(Index1)

    Table1_new = pd.DataFrame(columns=Columns,index=Index1)
    Table2_new = pd.DataFrame(columns=Columns,index=Index2)

    T1,T2=BuildObservationTable(Table1_new,Table2_new)
    return T1,T2

# Add a column in the table

def AddColumn(Table1,Table2):
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Columns=list(Table1.columns)

    Append=ColumnVariable(Table1,Table2)

    if Append!=None:
        Columns.append(Append)

        Table1_new = pd.DataFrame(columns=Columns,index=Index1)
        Table2_new = pd.DataFrame(columns=Columns,index=Index2)
        T1,T2=BuildObservationTable(Table1_new,Table2_new)

        return T1,T2

    else:
        return Table1,Table2

# Create a prefix closed set of strings from the given row headers

def PrefixClosed(label):
    Final=[]

    for i in range (len(label)):
        for j in range (len(ALPHABET)):
            temp=label[i]+ALPHABET[j]
            if (temp not in Final)&(temp not in label):
                Final.append(label[i]+ALPHABET[j])

    return Final

# Construct DFA from Dataframe

def DataframetToDfa(Table1,Table2):
    Index1=list(Table1.index)
    Index2=list(Table2.index)
    Index=Index1+Index2
    Columns=list(Table1.columns)

    Array1=np.array(Table1).tolist()
    Array2=np.array(Table2).tolist()
    Array=Array1+Array2

    LearnedTransition_test={}
    LearnedInitial_test=0
    LearnedFinal_test=[]
    LearnedStates_test=[]

    LearnedRows_test=[]
    LearnedStrings_test=[]
    row=0
    for i in range (len(Index1)):
        if Array1[i] not in LearnedRows_test:
            LearnedStates_test.append(row)
            row=row+1
            LearnedRows_test.append(Array1[i])
            LearnedStrings_test.append(Index1[i])

    for i in range (len(LearnedStates_test)):
        temp={}
        for j in range (len(ALPHABET)):
            String=LearnedStrings_test[i]+ALPHABET[j]
            I=Index.index(String)
            Row=Array[I]
            Append=LearnedRows_test.index(Row)

            temp[ALPHABET[j]]=Append

        LearnedTransition_test[i]=temp

    for i in range (len(LearnedStates_test)):
        for j in range (len(ALPHABET)):
            String=LearnedStrings_test[i]+ALPHABET[j]
            I=Index.index(String)
            Row=Array[I]
            State=LearnedRows_test.index(Row)
            Append=LearnedStrings_test[State]
            if (MembershipQuery(Append)==1)&(State not in
LearnedFinal_test)&(State in LearnedStates_test):
```

```
                LearnedFinal_test.append(State)


    LearnedDfa_test=[LearnedTransition_test,LearnedInitial_test,LearnedFinal_t
est,LearnedStates_test]

    return (LearnedDfa_test)

# Add Counter example and retrun the tables

def CounterExample(Table1,Table2):
    T1,T2=Table1,Table2

    while(1):
        T1,T2=AddRow(T1,T2)
        List=list(T1.index)

        if MembershipQuery(List[len(List)-1])==1:
            break

    return T1,T2

# Main

Table1,Table2=Initialize()
LearnedDfa=[]

while(1):

    while(CheckClosedness(Table1,Table2)!=True):
        Table1,Table2=AddRow(Table1,Table2)

    while(CheckConsistency(Table1,Table2)!=True):
        Table1,Table2=AddColumn(Table1,Table2)

    LearnedDfa=DataframetToDfa(Table1,Table2)

    if (EquivalenceQuery(LearnedDfa))!=True:
        Table1,Table2=CounterExample(Table1,Table2)
    else:
        break
```

```
print(LearnedDfa)
print(COUNT)
print(Table1)
print(Table2)
```

12.2 Appendix B: Simulink representation

The Simulink representation of the robot environment could not be taken as a code. However, screenshots of them have been attached in the report in FIGURE 6 in page 45 and the entire file is attached in the supplementary material. (please refer Figure 6: Simulation Overview in Simulink and Figure 7: State flow chart)