Implementing and Evaluating Automaton Learning Algorithms for a Software Testing Platform

Mohsen Khosravi Bakhtiari

Computer Science and Engineering, 30 credits

Halmstad 2015-02-20

Master thesis report

2015

Author: Mohsen Khosravi Bakhtiari
Supervisor: Karl Meinke
Examiner: Antanas Verikas

Mohsen Khosravi Bakhtiari

© Copyright Mohsen Khosravi Bakhtiari, 2015. All rights reserved.
School of Information Science, Computer and Electrical Engineering
Halmstad University
Acknowledgments

I am using this opportunity to express my deep gratitude to my supervisor, Prof. Karl Meinke, who spent a lot of time in supporting and helping me throughout the work at KTH University and to my examiner, Prof. Antanas Verikas, for his constructive comments and advices and also for his patience and great supports during my master’s studies at Halmstad University. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work.

I would also like to appreciate my dear friends at TSC Department for their great guidance throughout my work and for the great times I had with them in Fika (coffee break) times.

Last but not least, I would like to deeply thank my parents for their endless love, supports and encouragement during both my whole life and my studies in Sweden. I would like to wholeheartedly express my appreciation to both of you for giving me strength to chase my dreams. I would also like to appreciate my brother and sister who have been always supportive and encouraging to me in my whole life.
Abstract

The Software Reliability group at KTH-CSC has designed and built a novel test platform LBTest for black-box requirements testing of reactive and embedded software systems (e.g. web servers, automobile control units, etc). The main concept of LBTest is to create a large number of test cases by incorporation of an automata learning algorithm with a model checking algorithm (NuSMV). This platform aims to support different learned automata, learning algorithms and different model checking algorithms which can be combined to implement the paradigm of learning-based testing (LBT).

This thesis project investigates an existing published algorithm for learning deterministic finite automata (DFA) known as Kearns algorithm. The aim of this thesis is to investigate how effective Kearns algorithm is from a software testing perspective.

Angluin’s well-known L* DFA learning algorithm has a simple structure and implementation. On the other hand, Kearns algorithm has more complex, difficult structure and harder implementation than L* algorithm, however it is more efficient and faster. For this reason, the plan is to implement an advanced DFA learning algorithm, Kearns algorithm [4], from a description in the literature (using Java).

We consider a methodology to compare Kearns algorithm with Angluin’s DFA learning algorithm based on the master thesis of Czerny[8]. The comparisons between the Kearns and the L* algorithms are based on the number of membership and equivalence queries to investigate the difficulty of learning.
Contents

1 Introduction .......................................................................................................................... 1
  1.1 History of Learning......................................................................................................... 1
  1.2 Goals of this Thesis....................................................................................................... 1
  1.3 Reactive Systems.......................................................................................................... 2
  1.4 Black Box Testing......................................................................................................... 2
    1.4.1 Model Checking......................................................................................................... 2
    1.4.2 Requirements-based Testing (RBT).......................................................................... 3
    1.4.3 Model-based Testing................................................................................................ 3
    1.4.4 Learning-based Testing............................................................................................ 3
    1.4.5 Comparison of Different Testing Techniques.......................................................... 4
  1.5 Automata Learning........................................................................................................ 4

2 Background and Literature Survey .................................................................................. 7
  2.1 Automata Theory........................................................................................................... 7
    2.1.1 Definition (String).................................................................................................... 8
    2.1.2 Definition (Languages)........................................................................................... 8
  2.2 Deterministic Finite Automata...................................................................................... 8
    2.2.1 How the DFA processes strings?.............................................................................. 9
    2.2.2 Language of Automata............................................................................................ 11
  2.3 Applications of Finite Automata.................................................................................. 11

3 L* and Kearns Algorithms .............................................................................................. 15
  3.1 Active and Passive Learning......................................................................................... 15
  3.2 Exact Learning Using Queries...................................................................................... 15
  3.3 L* algorithm................................................................................................................ 16
  3.4 Suffix Trees for Kearns Algorithm [4].......................................................................... 18
    3.4.1 Exact Learning of Finite Automata........................................................................ 18
  3.5 Comparison L* and Kearns algorithms......................................................................... 22

4 Components for Kearns Algorithm for Learning DFA ............................................ 25
  4.1 Subroutine Sift(s, T)..................................................................................................... 25
  4.2 Tentative Hypothesis (T)................................................................................................ 26
  4.3 Procedure Update-Tree (y, T)..................................................................................... 27
    4.3.1 Least Common Ancestor....................................................................................... 27
  4.4 Processing of the Kearns Algorithm to Learn DFA.................................................. 28
    4.4.1 Equivalence Checker............................................................................................... 29

5 Experimental Results...................................................................................................... 33
  5.1 Experimental Method................................................................................................. 33
  5.2 How to create random DFA?...................................................................................... 33
  5.3 Experimental Setup Using Random DFA Generation.............................................. 34
  5.4 Comparison of the L* and Kearns algorithms with respect to difficulty of the learning considering membership queries......................................................... 35
  5.5 Comparison of the L* and Kearns algorithms with respect to difficulty of the learning considering equivalence queries......................................................... 40
6 Novelty of this Thesis ............................................................................ 47

7 Conclusion and Future Work ..................................................................... 49
  7.1 Conclusion .................................................................................................. 49
  7.1.1 Comparison of Membership Queries ......................................................... 49
  7.1.2 Comparison of Equivalence Queries ........................................................... 49
  7.2 Future Work .................................................................................................. 50

8 Appendix ........................................................................................................ 51

9 Bibliography .................................................................................................. 59
List of figures

Figure 1.1: LBTest Abstract Algorithm .......................................................... 4
Figure 1.2: Binary classification tree T .......................................................... 6
Figure 2.1: Deterministic Finite Automata ...................................................... 8
Figure 2.2: Finite automaton modelling an on/off switch ................................ 12
Figure 2.3: Finite automaton modelling of word "hello" ................................ 12
Figure 3.1: Components of the L* algorithm ................................................ 16
Figure 3.2: Target automaton M (DFA) ......................................................... 19
Figure 3.3: Sample execution of Kearns algorithm on target automaton M .... 21
Figure 4.1.a: The target automaton M. Figure 4.1.b: Sifting aaa on binary
classification tree T ......................................................................................... 26
Figure 4.2: Least Common Ancestor ............................................................... 28
Figure 4.3.a: Make an equivalence query on the target M. Figure 4.3.b: Make an
equivalence query on the hypothesis H ......................................................... 31
Figure 5.1: Behaviour of L* and Kearns algorithms respect to membership queries
achieved by using different r ........................................................................ 37
Figure 5.2: Average membership queries to correctly learn DFAs of different state
size with r = 50% ....................................................................................... 38
Figure 5.3: Average membership queries to correctly learn DFAs ................ 39
Figure 5.4: Behaviour of L* and Kearns algorithms respect to equivalence queries
achieved by using different r ........................................................................ 41
Figure 5.5: Average equivalence queries to correctly learn DFAs of different alphabet
size with r = 50% ....................................................................................... 42
Figure 5.6: Average equivalence queries to correctly learn DFAs of different state
size ............................................................................................................... 44
Figure 5.7: Average equivalence queries to correctly learn DFAs of different state size
r = 99% and 1% for Kearns and L* respectively ........................................ 45

List of tables

Table 4.1: Partition check table for target M and hypothesis H .................... 30
Table 8.1: r for 25, 50 and 75 state size ...................................................... 52
Table 8.2: Average of 15 executions for 25, 50 and 75 state size .................. 56
Table 8.3: Number of membership queries with different r in Kearns algorithms ... 57
Chapter 1

1 Introduction

In this chapter, first we will explain computational learning and introduce some efforts of computer scientists in this field. Then, we discuss the goals of this thesis, reactive systems and some types of testing techniques and their comparison. After that, we introduce LBT.

1.1 History of Learning

Humans throughout history have been studying and learning from their environment and this has led to new methods and ideas, learned from nature, in order to have a better life. This story continued until the human mind has thought to invent machines and algorithms and ways to train them from related information and data. To take an example, consider an abstract machine, also called an abstract computer, as a theoretical model of computer hardware or software systems used in automata theory. In the 1930’s, an abstract machine concept was investigated by A. Turing [1]. His goal was to find a way to distinguish what a computational machine is able to do and what it is not able to do. In the 1940’s and 1950’s, “finite automata” which are another kind of computational machine but simpler were first studied by computer scientists [1]. In the 1960s, Teslin [2] proposed automata models of learning systems for the first time and, in 1974, was recognized as automata learning (AL). The systems designed with automata learning were applicable to learn many complex algorithms. AL also could be generalized in some aspects in order to deal with other kinds of learning problems.

1.2 Goals of this Thesis

The Software Reliability group at KTH-CSC has designed and built a novel test platform LBTest [15, 16] for black-box requirements testing of reactive and embedded software systems (e.g., web servers, automobile control units, etc). The main concept of LBTest is to create a large number of test cases by incorporation of an automata learning algorithm with a model checking algorithm (NuSMV). This platform aims to support different learned automata, learning algorithms and different model checking algorithms which can be combined to implement the paradigm of learning-based testing (LBT). This paradigm has been pioneered in recent years by KTH-CSC researchers, and other testing groups around the world. It represents an important step forward in the automation of software testing [3].

This thesis project aims to investigate an existing published algorithm, Kearns algorithm [4], for learning deterministic finite automata (DFA), which would seem to be particularly effective from a software testing perspective. The plan is to implement this advanced DFA learning algorithm from a description in the literature.
Implementing and Evaluating Automaton Learning Algorithms for a Software Testing Platform

This algorithm must then be tested and benchmarked in isolation. If appropriate it could then be integrated into the LBTest platform. In combination with the currently integrated NuSMV model checker, the DFA learning algorithm could then be evaluated regarding the quality of learning-based testing which it supports.

1.3 Reactive Systems

Reactive systems are systems that react to the environment sequentially, through a series of inputs and outputs. During this procedure, performed in an iterative loop, these systems receive inputs from the environment and return the relevant outputs. In this field there are many examples including embedded systems such as cruise controllers in vehicles, system controlling mechanical devices such as train, air traffic control, medical devices or control systems in a nuclear reactor etc [3]. An important characteristic of this kind of system is that it is possible to model their behavior by automata. This in turn means we can automate testing with an automated testing method based on model checking and temporal logic.

Before we consider learning-based testing in more detail (section 1.4.4), it is appropriate to have a background in other testing techniques:

1.4 Black Box Testing

Various collections of testing techniques are available for the tester depending on whether the source code is accessible or not. If the source code is not available then the software is treated as a black box and testing techniques applied in this instance are called black box testing or functional testing [3].

In this section, we introduce some testing techniques. These testing techniques can be regarded as a form of black-box testing or conformance testing:

1.4.1 Model Checking

*Formal verification* is one of the methods used to assure the reliability of software systems. In this method, the correctness of the system is evaluated with respect to its accurately specified desired property, called a specification [9].

Over the past decades, model checking has emerged as one of the powerful and reliable methods to verify different types of systems against their specifications automatically e.g., communication protocols, digital circuits, reactive systems etc. Model checking aims to investigate the correctness of a desired property for a given model [3]. In other words, a model checker checks the correctness of models, represented as finite-state machines, against their specifications by exploring the state-space of those models [9]. Model checking also can be used as a black-box tool.
1.4.2 Requirements-based Testing (RBT)

In any engineering activity, including software engineering, formulating a problem using precise requirements is important. Solving a problem successfully has a direct relation to how well it is formulated. In computer science, one of the special formats used to describe a problem is called a requirements language. The requirements-based testing (RBT) task involves checking and evaluating that a set of requirements are true, complete, explicit and logically consistent, and also generate, from a black box viewpoint, a useful and adequate set of test cases from those requirements [11].

1.4.3 Model–based Testing

Model-based testing [10, 17] is the use of a design model for designing and executing the essential artifacts to implement software testing. The model can describe the behaviour of the System Under Test (SUT), that is generally abstract, considering test cases and their performance environment. Test cases are derived from these models (not from source code) thus MBT can be regarded as a form of black-box testing or conformance testing.

1.4.4 Learning-based Testing

Learning-based testing is an iterative method to automate specification-based black-box testing [3]. We can divide the framework of LBT in two parts as follows:

1. An SUT which is considered as black box and a formal specification for this SUT.
2. A learned model M of SUT.

All specification-based testing methods have the first part but the second one is a distinctive feature of LBT only. LBT can be defined as a heuristic iterative method. This method is based on the notion of learning a black-box SUT and it focuses on using tests as queries. Figure 1.1 shows the abstract algorithm for Learning-based Testing. We can describe the performance of this LBT algorithm as follows:

The LBT algorithm automatically generates a large number of test cases. If $\beta_1, \beta_2, ..., \beta_n$ are outputs achieved by implementation of n test case inputs $\alpha_1, \alpha_2, ..., \alpha_n$ on the SUT respectively, then the adaptive learning algorithm combines these $n$ pairs of input/output and integrates them into a learned model $M_n$. Then, the learned model $M_n$ is compared with the formal specification for the SUT by a model checker. If there is a contradiction, a counterexample is constructed by the model checker. The counterexample will be used as next input $\alpha_{n+1}$ for the SUT and after execution, the output $\beta_{n+1}$ is obtained. If SUT rejects this test case then the LBT algorithm terminates with a true negative. If it accepts test case, so $M_n$ was an
incorrect model and the test case was a false negative. In this case the LBT algorithm tries to build a refined model $M_{n+1}$.

![Figure 1.1: LBTest Abstract Algorithm.](image)

1.4.5 Comparison of Different Testing Techniques

All of the testing techniques introduced above can be considered as a form of black-box testing. If the source code is not available in these testing techniques then the software is treated as a black box and testing techniques applied in this instance are called black box testing or functional testing. All requirement-based, learning-based and model-based testing techniques generate a large number of test cases. The difference between them is that they use different methods to generate test cases. The requirement-based testing checks and evaluates that a set of requirements are true, complete, explicit and logically consistent, and generates a useful and adequate set of test cases from those requirements. The Learning-based testing generates a large number of test cases by combination of an automata learning algorithm with a model checking algorithm. Model-based testing is the use of a design model for designing and executing the essential artifacts to implement software testing. The model can describe the behaviour of the System Under Test (SUT), that is generally abstract, considering test cases and their performance environment. In model-based testing, test cases are derived from these models.

1.5 Automata Learning

Computational learning concerns modeling and designing algorithms to guess a target, a particular structure $m \in M$, from a set of possible structures $M$ given some information $D = d_1 \ldots d_i \ldots d_n$ about the target. This data and information are a series of data elements and the learner effort is to guess a reply for some invisible input in a cognizable artifact called a hypothesis or approximation $h \in H$ of the real system. Both $m$ and $h$ are able to receive input $d$ from sequence of data elements $D$. 
The learning algorithm implements supervised learning if a specific structure \( m \in M \) has well-known labels for data elements \( d_i \in D \) and the labels have been targeted to generate a hypothesis \( h \). On the other hand if the function values \( m(d_i) \) are unknown then there is an issue of recognizing the invisible structure from this unidentified data, called unsupervised learning.

According to the definitions above, exact learning is learning such that \( m(d_i) = h(d_i) \) for all \( d_i \in D \). This approach is used in order to learn our automata in this project.

In this thesis, we will consider computational learning algorithms that infer state machines. For this class of learning algorithm, also known as regular inference algorithms, the inputs are strings which are usually come from a formal language. In a generic regular inference algorithm, at the beginning, the learner does not have any information or knowledge about target \( M \). Therefore, the learner begins to ask queries to the teacher and oracle, which can be either membership or equivalence queries. The fundamental question for an equivalence query is whether the hypothesis \( H \) is equal to target \( M \) or not? If the answer is negative, then an oracle generates a counterexample which establishes the difference between \( H \) and \( M \).

In this project, Kearns algorithm is used for learning deterministic finite automata [4]. In order to consider its effectiveness for LBT we need to empirically analyse the run time performance of Kearns algorithm. We need to evaluate its performance on a large number of case studies. To achieve this goal, the first step is to generate a random DFA as a target automaton \( M \), with a user-defined number of states, transitions and final states. In the next step, the learner asks the teacher whether the initial state of target \( M \) is accepting or rejecting (membership query)? Then, the hypothesis automaton \( H \) is created with a single accepting or rejecting state, based on the answer of the membership query on \( M \), with self-loops labeled with all possible transitions. Again the learner asks a query but from the oracle (equivalence query) whether the hypothesis \( H \) is equal to target \( M \) or not? If the answer is negative then the oracle generates a counterexample which establishes the difference between \( H \) and \( M \).

In Kearns algorithm, a new way to classify input strings and their acceptance/rejection is introduced using a binary classification tree \( T \). The tree \( T \) is created by assigning internal nodes to a set of suffixes \( D \), and leaves to a set of prefixes \( S \). So, any string \( d \) in \( D \) is placed in each internal node and all strings \( s \) subset of \( S \) are placed in the left sub tree of the internal node provided that \( M \) rejects \( sd \), and in the other side of the sub tree (right side) when \( M \) accepts \( sd \). For example in the binary classification tree \( T \), Figure 1.2, suffixes \( \lambda , a, b \) are placed in internal nodes and prefixes ‘a’, ‘a’, ‘a’, ‘a’ are placed in leaves.
Kearns algorithm is executed in iterative phases. At the beginning of the iteration, the initialized binary classification tree $T$ and hypothesis $H$ are considered as the initial values $T_0$ and $H_0$ respectively. During each iteration, the learner discovers a new state of Hypothesis $H_i$ which is the last detected state of the target automaton $M$ (DFA) by checking $T_i$, using a least common ancestor method. Then, the equivalence checker (learner) poses an equivalence query to ask the oracle. The question is whether the hypothesis $H_i$ is equal to the target $M$ or not? If the answer is negative then the oracle generates a counterexample i.e. a finite input string on which the hypothesis and target differ. The equivalence checker’s function can be defined by an Advanced Depth First Search algorithm.

Finally, the tree will be updated and assumed as current $T_{i+1}$. This process will continue and when $M \equiv H_i$ so the learner has exactly learned the DFA.

Figure 1.2: Binary classification tree $T$.  

Implementing and Evaluating Automaton Learning Algorithms for a Software Testing Platform


Chapter 2

2 Background and Literature Survey

In this chapter we will describe some background concepts such as different related definitions, deterministic finite automata (DFA) and give a literature survey.

2.1 Automata Theory

Automata theory is a mathematical theory for the investigation of abstract machines and languages. In other words, automata theory studies abstract machines, defined as a theoretical model of a computer hardware or software systems, to solve the computational problems.

In the 1930's, the first abstract machine was investigated by A. Turing, from which the abilities and capabilities of today's computers have been taken. His goal was to find a way to distinguish what a computational machine is able to do and what it is not able to do. Nowadays, real machines are indebted to this abstract Turing machine.

In the 1940's and 1950's, "finite automata" which are another kind of machine but simpler were studied by computer scientists. These automata, invented to model brain function, became very useful for other several applications. Investigation of the formal grammar concept was the next step in this field, carried out by linguist N. Chomsky in the late 1950's. While they do not seem to be exactly a machine, formal grammars have close relations to abstract automata and are used as the basis of some software components, like some parts of compilers used today.

In 1969, the investigation of computational feasibility done by A.Turing was developed by S. Cook. The subject of study was which problems could and which could not be calculated efficiently. In other words, Cook could distinguish the solvable problems by computers from the ones that can in principle be solved [1].

It is noteworthy that these theoretical developments are the foundation of modern computer science and whatever the today's computers do has been originated from them. Nowadays, finite automata and formal grammars are involved in the construction and design of many of the major software. Turing also has helped to meet the expectations of our software types.

Now we look at some definitions such as string and languages related to automata theory:
2.1.1 Definition (String)

A string is a finite series of symbols selected from a set of alphabet symbols. To take an example, “ababbbab” is a string from the alphabet $\Sigma = \{a, b\}$. Also “aaaa” is another string chosen from $\Sigma$ [1].

2.1.2 Definition (Languages)

A set of strings all of which are selected from some $\Sigma^*$, where $\Sigma$ is a special alphabet, is termed a formal language. If $\Sigma$ is an alphabet and $L \subseteq \Sigma^*$, $L$ is a language over $\Sigma$ [1].

2.2 Deterministic Finite Automata

A finite automaton has a finite state set and its control shifts from one state to another in response to external inputs. One of the important differences between classes of finite automata is whether the control is deterministic, which means that it is impossible for the automaton to be in more than one state, or nondeterministic, meaning that it can be in several states at the same time. The term deterministic refers to the fact that there is just one transition for each input alphabet in each state [1, 18].

A deterministic finite automaton (DFA) is a finite automaton machine which is accepting or rejecting finite strings (Figure 2.1).

A DFA includes the following components:

- A finite set of states, conventional symbol: $Q$.
- The finite set of inputs, conventional symbol: $\Sigma$.
- A transition function takes a state and alphabet as an input and returns a
state. To take an example, if we suppose that \( q \) is one of the DFA’s states and also \( a \) is an input transition then the output result for state \( q \) and transition \( a \) is state \( p \). It is defined as a function such as:

\[
\partial (q,a) = p
\]

- The DFA should be initialized in a special state called a start state \( q_0 \) which could be any of the states in \( Q \).
- The final state set is a set of states that accepts the special strings; the conventional symbol is \( F \) which is a subset of \( Q \).

So from a mathematical point of view we can define a DFA \( A \) as follows:

\[
A = (Q, \Sigma, \partial, q_0, F).
\]

### 2.2.1 How the DFA processes strings?

It is really important to understand the process of accepting or rejecting the strings by DFA and how a DFA accepts a sequence of strings level by level. We suppose that the initial state is \( q_0 \) and the first input transition is \( a_1 \) from the set \( \Sigma = \{ a_1, a_2, a_3, \ldots a_n \} \) of input symbols. If the next state is \( q_1 \), then:

\[
\partial (q_0, a_1) = q_1
\]

In the next step, we can see that the next state is \( q_2 \) since we process the next input symbol, \( a_2 \), by evaluating function \( \partial \) with two inputs \( q_1, a_2 \):

\[
\partial (q_1, a_2) = q_2
\]

So we can continue to compute the next states \( q_3, q_4, q_n \):

\[
\partial (q_{i-1}, a_i) = q_i
\]

Finally, if \( q_n \in F \) (final states), then the DFA accepts the input string \( a_1a_2\ldots a_n \). Otherwise, it will be rejected by the DFA.

**Example:** In this example, the DFA accepts all and only the strings of \( a \)'s and \( b \)'s with a series of \( ab \) among the string. It is possible to write the language \( L \) in the following forms:

\[
W | W
\]

is one of the forms \( XabY \) for some strings \( X \) and \( Y \).

\[
(a|b)^* \quad ab \quad (a|b)^*\]

regular expression.

Here \( X \) and \( Y \) include \( a \)'s and \( b \)'s.
According to the description above, samples of strings in the language $L$ consist of $ab$, $bbaba$, $bbbbbbbbbabaaa$. Also, $\varepsilon$, $bbbbbbbbaaaaaa$, $bbbb$, $aaaaa$ are examples of strings which are not included in the language $L$.

What information do we need about an automaton that could accept the language $L$? The information about this DFA is:

- $\Sigma = \{a, b\}$.
- some sets of states, $Q$.
- initial state $q_0$.

In order to decide whether $ab$ is a substring of the input, the automaton has to consider the following conditions:

1. Has the automaton already seen $ab$? If this is true, then it would accept every series of further inputs; i.e., it does not change its position from now on.
2. Has the automaton never seen $ab$, but its most recent input was $a$, so if the automaton sees a $b$, it will have seen $ab$ and can accept everything it sees from here on?
3. Has the automaton never seen $ab$, but its last input was either non-existent (it just started) or it last saw a $b$? In this case, the automaton cannot accept until it first sees an $a$ and then sees a $b$ immediately after.

Every state can represent one of three conditions mentioned above. The start state is $q_0$ and condition (3) is represented by it. It is clear that when we are in the initial state, the automaton needs to see first $a$ and then $b$, so the string can be accepted by the automaton. But if we are in the initial state and the next input for seeing is $b$, then we are not able to see $ab$ at the moment and we must stay in this state (initial state):

$$\partial (q_0, b) = q_0.$$

Also, if we are in the initial state $(q_0)$ and an $a$ is the next character as an input symbol, then condition (2) holds. It means that we have never seen $ab$ but $a$ has already seen. We assuming that our transition from $q_0$ on input $a$ is:

$$\partial (q_0, a) = q_2.$$ 

Finally, if the algorithm sees an $a$ but from state $q_2$, we are not in a clear situation because $ab$ has not seen by automaton, but $a$ was the last symbol and the automaton can still wait for a $b$. If the transition function $\partial$ would be:

$$\partial (q_2, a) = q_2,$$

then the automaton is waiting to receive input $b$ and the equation $\partial (q_2, b) = q_1$ can accept $b$ by transition from $q_2$ on input $b$ which corresponds to condition (1).
The last state which should be considered is $q_1$. We have already seen $ab$ so $q_1$ should be final state and there are two transition functions for it:

$$\partial(q_1, a) = \partial(q_1, b) = q_1.$$ 

Thus, the automaton $A$ that accepts the language $L$ of strings with a substring $ab$ should be defined by:

$$A = (\{q_0, q_1, q_2\}, \{a, b\}, \partial, q_0, \{q_1\}).$$

### 2.2.2 Language of Automata

An automaton accepts or rejects strings. A string will be accepted if it starts to be processed from the starting state and the first symbol of the string causes the first transition from starting state to the next state. This must continue until it leads to an accepting state.

In terms of a transition diagram, a string will be accepted by an automaton if the label of the path (transitions) from initial state to some final states is the same as string [1, 20, 21].

### 2.3 Applications of Finite Automata

As mentioned before, many complex and important kinds of hardware and software use finite automata as useful model form, for example:

1. Software for designing and checking the behaviour of digital circuits.
2. The “lexical analyser“ of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords and punctuation.
3. Software for scanning large bodies of text, such as collections of web pages to find occurrences of words, phrases or other patterns.
4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange of information.

We can find many systems and components, such as those mentioned above, that may be observed as being in just one of the limited numbers of “states” at all times. Actually, the role of a state in finite automata is to maintain the related section of the system's history. Since the number of states is limited, registering the whole part of the history is almost impossible. Therefore, in designing the systems, it is very important to know which part is useful and which part can be skipped. One of the best benefits of using finite automata is having a limited number of states, causing the systems to run with the certain fixed set of resources. For example, it is possible
to implement it in hardware as a circuit or as a part of a specific program, searching for limited amount of data from entire data or many kinds of decisions like these.

**Example 1.1:** This example observes the simplest finite automaton, called an on/off switch. There are two states of on and off for device, which means that the device can recognize whether switch is in the on or off state. Also, the user has two possibilities (on/off) for changing the states and the operations of the system depend on the position of the switch. In other words, if the switch is in the off state, the user can change it to on by pushing the switch button and vice versa.

![Finite automaton modelling an on/off switch.](image)

The finite automaton model for the switch is shown in Figure 2.2. The circles demonstrate the states in the finite automata. In example 1.1, there are two states named on and off. Also arcs between states are tagged by push which expresses external influences on the system. If the switch is off, when the user pushes the button it can be imagined that the state off is directed to the state on by push transition and vice versa. One of the states should be chosen as the start state, the place where the system is initialized.

In this example, the switch off is specified as the start state and is named Start. It is necessary to have at least one or more final states to show that the inputs are applicable in some way. For example, we can consider the on state as an accepting state to control the machine. In order to indicate the final state, it is conventional to show it by double circle.

**Example 1.2:** Let’s take another example more complex than above. A lexical analyser has been shown in figure 3, modelled by another finite automaton. The task of this automaton is to detect the keyword “hello”. It can be represented by six states, each state represents a different position in the word “hello” that has been reached so far. The process starts from empty state, called the initial state, and it will continue until the complete word “hello” has been detected.

![Finite automaton modelling of word “hello”.](image)

In figure 2.3, the six states include the prefixes of the word “hello” from the empty string to the complete word. Inputs correspond to letters. In this example, the lexical analyser checks out one symbol of the string that it is compiling at the time, and the
next symbol to be considered is the next input to the automaton. There is just one final state because the task of the lexical analyser is finished when “hello” has been discovered [19].
Chapter 3

3 L* and Kearns Algorithms

3.1 Active and Passive Learning

There are some special conditions in which it is expensive to tag unlabeled data manually. In such a case, a learning algorithm is able to query a teacher for labels. This kind of supervised learning is named active learning. In other words, active learning is a particular instance of supervised machine learning in which a learning algorithm is able to ask queries to the user to achieve the related outputs.

In active learning, we model experimentation by giving the learner permission to make membership queries: When the target concept \( c \) is being learned, the learner by having access to an oracle for any input \( x \) will receive the appropriate target \( c(x) \). Therefore, the learning algorithm examines special inputs then it considers their target classification rather than receiving random labeled inputs.

In the passive model, a learning algorithm has no control over the sample set and it receives responses from the environment without being able to submit a query to a teacher about some data points. Therefore, the environment is ineffective for the learner and effectively provides only random samples of the target concepts.

3.2 Exact Learning Using Queries

One of the learning models used in computational learning theory is exact learning using queries in order to learn the target concept \( c \).

In the passive model, the learned model is acceptable to within a close similarity to the target. But in the case of exact learning, the aim is to have output exactly equal to the target concept \( c \). In this case, the learner is not confined to the random examples and has access to oracles in order to respond to two kinds of queries:

- **Membership queries:**

For a membership query while learning the target concept \( c \), the learning algorithm has access to a teacher. Thus, for any instance \( x \) chosen by the learner as an active query the teacher responds with a correct classification \( C(x) \) as output.

- **Equivalence queries:**

The learning algorithm tries to learn an unknown target from a well-known concept class \( C \). It presents a hypothesis concept \( h \), the output of the algorithm, in order to exactly represent the target.
The learner can be successful in learning the correct target if \( h(x) = c(x) \) for all instance \( x \). Otherwise, the learner will receive feedback, called a counterexample, which is some input value \( x \) such that \( h(x) \neq c(x) \).

### 3.3 L* algorithm

The L* Algorithm is a well-known active learning for DFA introduced by Dana Angluin [5] and described by Chen [6] and Berg [7]. The goal of the L* algorithm is to collect enough information about an unidentified system, modelled as deterministic finite automata, to construct a hypothesis automaton as a DFA by asking queries from a teacher, called membership and equivalence queries.

The L* algorithm has several components shown in Figure 2.4. In this algorithm, the learner is responsible for controlling the learning procedure by choosing appropriate queries to ask the teacher. On the other hand, the teacher answers the queries supplied by the learner based on information from the system under test (SUT), modelled as an automaton. Finally, all information achieved by the learner is memorized in an observation table.

If a DFA \( A = (Q, \Sigma, \delta, q_0, F) \) is the result of modeling an unknown system, the implicit learner task is recognizing the accepted language \( L(A) \). A simple mapping of strings, actual observations, is created from the input alphabet \( \Sigma \) to a set of \{accepting, rejecting\} based on accepting or rejecting the input alphabet by automaton \( A \).

At the beginning, the learner does not have any information about the DFA \( A \). Therefore, it needs to make equivalence and membership queries in order to gather missing information and send them to a teacher.

**Figure 3.1:** Components of the L* algorithm.

**Observation table:**

Angluin’s L* algorithm saves collected information in an observation table \( \mathcal{O}T = (P, S, T) \) for a given alphabet \( \Sigma \) such that:
a) \( P \subseteq \Sigma^* \) is a non-empty finite prefix-closed set. A set is prefix-closed if and only if every prefix of every member of the set is also a member of the set.

b) \( S \subseteq \Sigma^* \) is a non-empty finite suffix-closed set. A set is suffix-closed if and only if every suffix of every member of the set is also a member of the set.

c) A finite function \( T \) mapping \( (\{ P \cup P \} \cdot \Sigma) \) to \{Accept, reject\}.

An observation table \( OT \) can be represented as a 2-dimensional array with rows tagged by the strings \( (P \cup P \cdot \Sigma) \) and columns tagged by \( S \). The first part of the observation table \( OT \) is indexed by \( P \) and the second part is indexed by all strings which have been observed for the first time and they did not process in the first part of the observation table \( OT \) and are of the form \( pa \) such that: \( p \in P \) and \( a \in \Sigma \). The table is indexed column-wise by a suffix-closed set \( S \) of strings. The function \( T \) maps a row label \( p \in P \) and a column \( s \in S \) to the set \{Accept, Reject\}. If \( ps \in L(M) \) then the entry field corresponding to that row label and column label is Accept, otherwise it is Reject.

For every \( p \in (P \cup P \cdot \Sigma) \), a function \( row(p) \) denotes a finite function from \( S \) to \{Accept, Reject\}. In other words, \( row(p) \) expresses the tuple of entries in the observation table \( OT \) for row tagged \( P \). The states of the hypothesis for DFA are determined by all distinct rows of the form \( row(p) \) where \( p \in P \). The hypothesis DFA can be discovered from the observation table using the rows of entries tagged by element of \( P \cdot \Sigma \) to create the transition function.

In order to construct the DFA, two criteria should be fulfilled by observation table \( OT \):

1) Closed: An observation table \( OT \) is closed if for each \( p \in P \cdot \Sigma \) there exists a \( p' \in P \) such that \( row(p') = row(p) \).

2) Consistent: An observation table \( OT \) is consistent if whenever \( p \) and \( p' \in P \) such that \( row(p') = row(p) \) then for all \( \alpha \in \Sigma \), \( row(p'. \alpha) = row(p. \alpha) \).

When the observations are closed and consistent, the learner can generate a hypothesis of DFA \( A = (Q, \Sigma, \delta, q_0, F) \) as follows:

- \( Q = \{row(p) \mid p \in P\} \), note: the set of distinct rows,
- \( q_0 = row(\epsilon) \),
- \( F = \{row(p) \mid p \in P \text{ and } T(p) = \text{Accept}\} \),
- \( \delta (row(p), \alpha) = row(p. \alpha) \).

The \( L^* \) algorithm memorizes the observation table \( OT \). The sets \( P \) and \( S \) are both initialized to \{\( \epsilon \)\}. After that, \( L^* \) algorithm makes membership queries on each \( \alpha \in \Sigma \) and \( \epsilon \), which result is either an Accept or Reject for each query. Then, the \( L^* \) algorithm should check that \( OT \) is closed and consistent.
If $\mathcal{OT}$ is not consistent, then inconsistency is resolved through finding two strings $p, p' \in P, \alpha \in \Sigma$ and $s \in S$ such that $\text{row } (p') = \text{row } (p)$ but $T(pa, s) \neq T(p'a, s)$ and the new suffix $\alpha s$ should be added to $S$ and by asking membership queries the algorithm fills the missing entries in the new columns.

If $\mathcal{OT}$ is not closed then the $L^*$ algorithm finds $p \in P$ and $\alpha \in \Sigma$ such that $\text{row } (pa) \neq \text{row } (p^*)$ for all $p^* \in P$ and adds $pa$ to $P$. The missing entries in $\mathcal{OT}$ are filled through membership queries [3].

When $\mathcal{OT}$ become closed and consistent then the hypothesis $H$ can be created and checked by an equivalence checker for correctness against the target $M$, using an equivalence query to the oracle. If the answer for equivalence query is positive then $L^*$ algorithm has exactly learned the DFA. Otherwise, the learner will receive a counterexample $\gamma$, such that $\gamma \in \mathcal{L}(M) \Leftrightarrow \gamma \notin \mathcal{L}(H)$.

This process will continue until the equivalence checker does not create a counterexample.

3.4 Suffix Trees for Kearns Algorithm [4]

In this section we give an abstract view of Kearns algorithm using classification trees introduced in section 1.5. Kearns learning algorithm for DFA is an active learning algorithm. Therefore we begin by carefully defining what we mean by active learning.

3.4.1 Exact Learning of Finite Automata

According to the above concepts for learning algorithms, it is now time to start describing Kearns exact active learning algorithm which learns a deterministic finite automaton from membership and equivalence queries.

If we suppose $M$ be a target automaton (Figure 3.1), the aim of Kearns algorithm is to discover new states of $M$ step by step.

The algorithm executes in steps. In each step, it generates a new automaton named the tentative hypothesis $H_i$ whose states are the last detected states of target $M$. Then, it computes an equivalence query on $H_i$ and $M$. The generated counterexample from this equivalence query causes the algorithm to execute membership queries to recognize what the next new state is. If all states of target $M$ are discovered, $M \equiv H_i$ and the algorithm has successfully learned target $M$. 

• **Access strings (prefixes) and distinguish strings (suffixes):**

The algorithm generates a new automaton in each step named the tentative hypothesis $H_i$ whose states are the last detected states of target $M$. But the question is how does the learning algorithm have access to the information regarding the states of $M$? The answer is: the algorithm stores a set $S$ of access strings (prefixes) and a set $D$ of distinguishing strings (suffixes) in order to gather the information about the states.

$M[s]$ is defined as an access string if:

For each string $s \in S$, it starts to process from an initial state and leads to one of the other states of $M$.

If we suppose that $s$ and $s'$ are two different strings, we can say that a distinguishing string is a string $d$ such that one of $sd$ or $s'd$ is accepted and the other one is rejected by the automaton.

• **Classification tree:**

The algorithm should memorize access and distinguishing strings, which is the sum total of all information about the states of $M$. The best method to classify prefix strings is to use a binary classification tree. The structure of the classification tree for this algorithm is assigning internal nodes to suffixes, and leaves to prefixes. Any string $d \in D$ is placed in each internal node and all strings $s \in S$ are placed in the left subtree of the internal node provided that $M$ rejects $sd$, and in the right subtree when $M$ accepts $sd$.

Figure 3.2 shows how the Kearns algorithm learns the DFA (Figure 3.1). The algorithm runs in four steps in this example.
The first column defines the hypothesis $H_i$ whose states are discovered by updating the binary classification tree $T_i$ using a counterexample generated by the equivalence checker. After each update, a prefix is placed in a new leaf node of the tree $T_i$, a new discovered state of DFA. The second column demonstrates the corresponding binary classification tree $T_i$.

Kearns algorithm can learn each one of the states of target automaton $M$ and at the end of execution all states are detected, shown in the last step (step four) of the execution in Hypothesis $H_4$. Finally, if we compare hypothesis $H_4$ and the target automaton $M$, they are equivalent.
Chapter 3. L* and Kearns Algorithms

Hypothesis H

$H_1$:

Binary classification tree $T$

$T_1$:

$H_2$:

$T_2$:

$H_3$:

$T_3$:

$H_4$:

Figure 3.3: Sample execution of Kearns algorithm on target automaton $M$. 
Implementing and Evaluating Automaton Learning Algorithms for a Software Testing Platform

3.5 Comparison of the L* and Kearns Algorithms

Active learning is a particular instance of supervised machine learning in which a learning algorithm is capable of querying the teacher to achieve the related outputs. While, in the passive model, a learning algorithm has definitely no control over the sample of labelled examples drawn and provides only random examples of the target concepts. The L* and Kearns algorithms are typical of active learning. These exact active learning algorithms learn a deterministic finite automaton from membership and equivalence queries.

The Kearns algorithm starts the learning process by making a membership query on initial state λ to find out if the target automaton M accepts or rejects λ. The learner creates an initial hypothesis H including a single (accepting or rejecting based on the result of making a membership query on initial state λ) state labelled with λ and self-loop transitions for all input symbols of the target M. Then, the learner makes an equivalence query on H and target M. If there is a contradiction, a counterexample γ is generated by equivalence checker. In the next step, the Kearns algorithm initializes a binary classification tree T with the first internal node (root) tagged with the distinguishing string λ and also two leaves tagged with access string λ and γ. The initialized binary classification tree T and hypothesis H are considered as the initial values T₀ and H₀ respectively. After the initialization, Kearns algorithm is executed in iterative phases. During each iteration, a new tentative hypothesis Hᵢ is generated whose states are detected from the binary classification tree Tᵢ. The learner performs an equivalence query on hypothesis Hᵢ. If M ≡ Hᵢ, then the Kearns algorithm could be successful in learning the target M. Otherwise, γᵢ detected from implement of equivalence checker on hypothesis Hᵢ is a counterexample. Finally, the tree will be updated and assumed as current Tᵢ₊₁. This process will continue and when M ≡ Hᵢ so the learner has exactly learned the target M. On the other hand, in L* algorithm an observation table OT can be represented as a 2-dimensional array with rows tagged by the strings (P ∪ P.∑) and columns tagged by s ∈ S ⊆ ∑∗. The observation table OT can be divided into a BLUE and a RED part as described in [22]. The RED part includes all rows to represent states in the generated hypothesis Hᵢ. The BLUE part contains rows with information about all successor states to which a transition from a state in the RED part could arrive BLUE = (RED · ∑) \ RED. First, the observation table OT is initialized according to the current input alphabet by learner, such that the empty string ε is added to the RED and all input alphabets β ∈ ∑ are added to the BLUE part. The suffixes are initially set to S = {ε}. Then, the learner makes the membership queries on all unknown row entries. After the initialization, L* algorithm is executed in iterative phases. During each iteration, the learner should check that the observation table OT is closed and consistent. If the observation table OT is closed and consistent, the learner is able to build a hypothesis Hᵢ from the observation table OT and computes an equivalence query on Hᵢ and target M. If there
is a contradiction, $\gamma_i$ is the generated counterexample from this equivalence query. The counterexample $\gamma_i$ should be added to the observation table $\mathcal{O}T$. Therefore, all prefixes and counterexample $\gamma_i$ have to be moved or added to the RED part and all successive prefixes are added to the BLUE part if they are not in $\text{BLUE} \cup \text{RED}$. Then, the learner makes the membership queries on all new row entries. If $\mathcal{O}T$ is not closed or consistent, the learner is not able to build a hypothesis $H_i$ from $\mathcal{O}T$ and discover a new state of target $M$. Therefore, L* algorithm tries to establish closed or consistent in observation table for the next iteration.

If we compare two learning algorithms, in each iteration, the Kearns algorithm generates a new automaton named the tentative hypothesis $H_i$ whose states are the last detected states of the target $M$. Thus, Kearns algorithms can discover a new state of target $M$. While, the L* algorithm should check that the observation table $\mathcal{O}T$ is closed and consistent. If $\mathcal{O}T$ is not closed or consistent, the learner is not able to build a hypothesis $H_i$ from $\mathcal{O}T$ and discover a new state of target $M$. Thus, L* algorithm tries to establish closed or consistent in observation table for the next iteration. The run time for each performance of an equivalence query really depends on when the observation table is closed and consistent, this time is variable.

As we already explained, Kearns algorithm is executed in iterative phases. Each iteration requires the processing of a counterexample to update the binary classification tree $T_i$ and creates a new tentative hypothesis $H_i$ whose states are detected from the updated binary classification tree $T_i$. A counter example of length $n$ requires at least $n$ membership queries. Thus, the number of membership queries used in Kearns algorithm is directly related to the length of the counterexamples. While, in each iteration of the L* algorithm, when the observation table $\mathcal{O}T$ is closed and consistent, the learner is able to build a hypothesis $H_i$ from the observation table $\mathcal{O}T$. Then, equivalence checker computes an equivalence query on $H_i$ and target $M$. The generated counterexample from this equivalence query is $\gamma_i$. The counterexample $\gamma_i$ should be added to the observation table $\mathcal{O}T$. Therefore, all prefixes and counterexample $\gamma_i$ have to be moved or added to the RED part and all successive prefixes are added to the BLUE part if they are not in $\text{BLUE} \cup \text{RED}$. Then, the learner makes the membership queries on all new row entries. In conclusion, the number of membership queries used in L* algorithm depends on the number of counterexamples, new prefixes and suffixes.

In each execution, Kearns learning algorithm can discover a new state of the target $M$ by having access to the oracle in order to answer an equivalence query. Each iteration requires the processing of a counterexample to update the binary classification tree $T_i$ and creates a new tentative hypothesis $H_i$ whose states are detected from the updated binary classification tree $T_i$. So, the number of equivalence queries is independent of the alphabet size. While, in L* algorithm, a higher number of input alphabets leads to making a more complex representation of the target $M$. Thus, there are more information to learn for the L* algorithm. This behaviour might be referred to the properties of the observation table $\mathcal{O}T$. The learner needs to
observe more behaviour. On the other hand, the learner only makes an equivalence query if the observation table is closed and consistent. In conclusion, the number of equivalence queries is dependent on the alphabet size.
Chapter 4

4 Components for Kearns Algorithm for Learning DFA

In this chapter we give a detailed description of the components of Kearns algorithm, building on the abstract description of Kearns algorithm using suffix trees given in Chapter 3.

In each iteration, Kearns algorithm has access to oracles to answer membership and equivalence queries. So, the learner discovers a new state for the hypothesis $H_i$ which is the most recently detected state of the target automaton $M$. If $M \neq H_i$, there must be a counterexample to drive state detection. The process will continue and when $M \equiv H_i$, the learner could exactly learn the DFA.

Four main subroutines are needed by Kearns algorithm. These are described as follows:

4.1 Subroutine Sift($s$, $T$)

In each iteration, a subroutine Sift is used to find the new states of the target $M$ by Procedure Update-tree ($\gamma$, $T$) and create a hypothesis $H$ by Procedure Tentative Hypothesis ($T$).

In subroutine Sift, the input is a string $s$ and the current binary classification tree $T$ and the output is one of the access strings of $T$: the subroutine Sift considers the root of the binary classification tree $T$ as the current node. If the current node is an internal node labelled by the distinguishing string $d$, the subroutine Sift makes a membership query on the string $sd$ and the current node moves to the left or right depending on the answer of membership query (left for reject, right for accept). This process will continue until the current node is a leaf (access string) and it would be the output of subroutine Sift.

Sift($s$, $T$):

- Initialization: consider the root of the suffix tree $T$ as the current node.
- Main loop:
  1. At the current node of $T$, consider its distinguishing string $d$.
  2. Do a membership query on $sd$. If the target $M$ accepts it, move the current node from the current position to the right child of current node. Otherwise, move the current node from the current position to the left child of current node.

Note: All internal nodes in $T$ are assigned distinguishing strings.

Note: The current node works like pointer.

- 3. If the current node is a leaf node of $T$, return the access string saved at
Implementing and Evaluating Automaton Learning Algorithms for a Software Testing Platform

... this leaf.
Otherwise, repeat the main loop.

Figure 4.1.b shows the result of Sifting \( (aaa, T) \) when considering the target automaton \( M \) in Figure 4.1.a. The result of sifting \( aaa \) on the binary tree \( T \) has been signed with a dashed circle.

![Diagram of automaton and binary tree]

Figure 4.1.a: The target automaton \( M \).  
Figure 4.1.b: Sifting \( aaa \) on binary classification tree \( T \).

### 4.2 Tentative Hypothesis (\( T \))

The next procedure is known as Tentative hypothesis \( (T) \) used in Kearns algorithm to create the hypothesis \( H \) from a suffix tree \( T \). Tentative hypothesis \( (T) \) starts to build states for hypothesis \( H \) corresponding to all the access strings of \( T \). The subroutine \textit{Sift} is used in the body of Tentative hypothesis \( (T) \) to make the transition for each created state.

**Tentative Hypothesis \( (T) \):**

- Each state \( q \) of \( H \) is equal to an access string of \( T \). Build states for hypothesis \( H \) corresponding to all the access strings of \( T \). Consider the empty access string \( \lambda \) as start state \( q_0 \).
  
  Note: the number of states of \( H \) = the number of access strings of \( T \).

- For each input symbol \( b \in \Sigma \), make the \( b \)-transition for each state which has been created for \( H \) by:
  1. \( s' \leftarrow \text{Sift}(sb, T) \).
  2. Make the \( b \)-transition from state \( s \) to \( s' \).

  Note: \( b \) is a subset of \{alphabets\).
4.3 Procedure Update-Tree ($\gamma, T$)

One of the most complex and important parts of Kearns algorithm is updating the suffix tree $T$, because in this process the next state of the target automaton $M$ will be discovered. In Procedure Update-tree ($\gamma, T$), the inputs are the current binary classification tree $T$ and a counterexample string $\gamma$ which is created from making an equivalence query between the current hypothesis $H$ and the target $M$. The output is a new access string that will be the next discovered state in target $M$.

Procedure Update-tree ($\gamma, T$):

- For each prefix $\gamma[i]$ of $\gamma$ | $\gamma[i] = \gamma_1...\gamma_i$:
  1. $s_i \leftarrow \text{Sift} (\gamma[i], T)$.
  2. $\hat{s}_i = H[\gamma[i]]$.
- Let $1 \leq j \leq |\gamma|$ be the least $i$ such that $s_i \neq \hat{s}_i$.
- Replace the leaf node tagged with the prefix $s_{j-1}$ in $T$ by an internal node with two leaf nodes (prefixes). Assign access string $s_{j-1}$ to one leaf node and the new access string $\gamma[j-1]$ to the other one. The distinguishing string $\gamma_d$ is the tag for new internal node. It should be noted that $s_j$ and $\hat{s}_j$ are distinguished by $d$ which can be computed by the Least Common Ancestor algorithm, described next.

4.3.1 Least Common Ancestor

In graph theory, the least common ancestor (LCA) of two nodes $s_j$ and $\hat{s}_j$ in a tree $T$ is the deepest node $d$ that is common ancestor for both $s_j$ and $\hat{s}_j$. In other words, when each node is directed to its parent, the LCA can be quickly specified by finding the first junction of the paths from $s_j$ and $\hat{s}_j$ to the root. Least Common Ancestor algorithm finds the internal node $d$, which is the least common ancestor of pair leaf nodes $s_j$ and $\hat{s}_j$. In Figure 4.2 if $s_j = 'a'$ and $\hat{s}_j = 'aa'$, the LCA is $\lambda$. 

- Return $H$. 

- For each prefix $\gamma[i]$ of $\gamma$ | $\gamma[i] = \gamma_1...\gamma_i$:
  1. $s_i \leftarrow \text{Sift} (\gamma[i], T)$.
  2. $\hat{s}_i = H[\gamma[i]]$.
- Let $1 \leq j \leq |\gamma|$ be the least $i$ such that $s_i \neq \hat{s}_i$.
- Replace the leaf node tagged with the prefix $s_{j-1}$ in $T$ by an internal node with two leaf nodes (prefixes). Assign access string $s_{j-1}$ to one leaf node and the new access string $\gamma[j-1]$ to the other one. The distinguishing string $\gamma_d$ is the tag for new internal node. It should be noted that $s_j$ and $\hat{s}_j$ are distinguished by $d$ which can be computed by the Least Common Ancestor algorithm, described next.

- For each prefix $\gamma[i]$ of $\gamma$ | $\gamma[i] = \gamma_1...\gamma_i$:
  1. $s_i \leftarrow \text{Sift} (\gamma[i], T)$.
  2. $\hat{s}_i = H[\gamma[i]]$.
- Let $1 \leq j \leq |\gamma|$ be the least $i$ such that $s_i \neq \hat{s}_i$.
- Replace the leaf node tagged with the prefix $s_{j-1}$ in $T$ by an internal node with two leaf nodes (prefixes). Assign access string $s_{j-1}$ to one leaf node and the new access string $\gamma[j-1]$ to the other one. The distinguishing string $\gamma_d$ is the tag for new internal node. It should be noted that $s_j$ and $\hat{s}_j$ are distinguished by $d$ which can be computed by the Least Common Ancestor algorithm, described next.
4.4 Processing of the Kearns Algorithm to Learn DFA

We define the main routine used in Kearns algorithm as follows:

**Step 1:**
- initialization:
  1. Make a membership query on initial state $\lambda$ to find out if the target automaton $M$ accepts or rejects $\lambda$.
  2. Create an initial hypothesis $H_0$ including a single (accepting or rejecting) state labelled with $\lambda$ and self-loop transitions for all input symbols of target $M$.
  3. Make an equivalence query on $H_0$ (using Advanced Depth First Search as an equivalence checker); name the counterexample $\gamma_0$.
  4. Initialize the binary classification tree $T_0$ with the first internal node (root) tagged with the distinguishing string $\lambda$ and also two leaves tagged with access string $\lambda$ and $\gamma_0$.

**Step 2:**
- Main loop:
  1. Assume $T_i$ as a current classification tree.
  2. $H_i \leftarrow$ Tentative-Hypothesis $(T_i)$.
  3. Make an equivalence query on hypothesis $H_i$. If $M \equiv H_i$, then output $H_i$ and print “the target automaton $M$ is equal to hypothesis $H_i$”. Otherwise, $\gamma_i$ detected from implement of equivalence checker on hypothesis $H$ (using Advanced Depth First Search equivalence checker) is a counterexample.
  4. Update-tree $(T_i, \gamma_i)$.
  5. Repeat Main loop.
4.4.1 Equivalence Checker

In each iteration of Kearns algorithm, one new state of the target automaton \( M \) must be discovered. For this, an Equivalence Checker (EC) is used to equivalence check whether \( M \equiv H \) or not? One of the methods in order to check the two automata is DFS (Depth First Search). Therefore, this equivalence checker uses two DFSs, with the same performance, that work on each of automata \( M \) and \( H \) parallel and concurrently.

Let’s explain the DFS method run on each automaton \( M \) and \( H \) at each step. The DFS starts from the initial state and makes a membership query on it to check whether the input alphabet, for example \( a \), is accepted by the target automaton or not? Then the state will be saved as a visited state in a stack memory. In each step of the EC run, if both \( M \) and \( H \) accept or reject the input alphabet, then the process will continue until all states of automaton have been visited, otherwise when one of the \( M \) or \( H \) accepts and the other rejects the input alphabet equivalence checker says that two automata are not equal and there is a counterexample. In order to find the counterexample we also need another stack memory to save transitions at each step.

**Note:** For each state, all transitions in automaton should be checked.

**Partitioning target automaton \( M \) and hypothesis \( H \):**

If we consider target automaton \( M = (Q, \sum, \delta, q_0, F) \) and hypothesis \( H = (P, \sum, \delta', p_0, F') \), \( q_i \in Q \) and \( p_j \in P \) are equivalent:

If \( \delta(q_i, a) \in F \) and \( \delta(p_j, a) \in F' \) or if \( \delta(q_i, a) \notin F \) and \( (p_j, a) \notin F' \). Then \( q_i \) and \( p_j \) are in the same equivalence class and they are placed in the same partition.

**Problem in DFS:**

Sometimes, after several runs of the Equivalence Checker (EC) on the target automaton \( M \) and hypothesis \( H \), all states are visited but still there is a counterexample that needs more than one time exploring of visited states to be discovered. In this case, the normal DFS does not work. Therefore, an advanced DFS is used which has solved the problem of finding the counterexample.

**What is advanced DFS?**

In the simple DFS method, we are not able to check the observed states for more than one time and sometimes this equivalence checker is not able to find a counterexample after investigating all states. On the other hand, we still need to discover a new state of \( M \). For this reason, we built and designed an advanced DFS method with ability of checking the observed states of automata for more than one time. In this method, the EC uses two advance DFSs, with the same performance, that work on each of automata \( M \) and \( H \) parallel and concurrently. EC starts the second time of checking if the following conditions exist together: 1) EC is not able to find a counterexample after investigating all states. 2) Current state in the target \( M \) is visited by equivalence checker and it is not in the same partition with the current state in \( H \). Then, EC will continue checking states until \( M \) or \( H \) accepts the input
alphabet and another one rejects it. If the transition in each run is saved in a stack memory, finally the counterexample is found by the EC.

Example 4.1: the target automaton $M = \{(q_0, q_1, q_2, q_3, q_4), \{a, b\}, \delta, q_1, (q_3)\}$ is a DFA in Figure 4.3.a and Figure 4.3.b is the hypothesis $H = \{(\lambda, a, aa), \{a, b\}, \delta, \lambda, \{aa\}\}$. The equivalence checker checks two automata and if they are not equal, it returns a counterexample. In this example, according to the Table 4.1(output state column, row 1...5), all states of automaton $M$ have been checked. The current visited state is $q_0$ (input state column, row 6) in target $M$. Equivalence checker needs to explore $q_2$, it already has visited this state, more than one time to find a new counterexample. In Table 4.1 (row 6 in target $M$ and hypothesis $H$), state $q_2 = \delta(q_0, a)$ is not in the same equivalence class with state $aa = \delta(a, a)$ because the target automaton $M$ rejects the input alphabet $a$, while the hypothesis $H$ accepts it. Therefore, the equivalence checker will continue investigating the states in the same method. The stopping condition for the algorithm is when one of $M$ or $H$ accepts the input alphabet and the other one rejects it. In this example, counterexample $\gamma = aaabaa$ is generated by the EC.

Code in Appendix shows the function of Advanced DFS. It first tries to find a counterexample by making membership queries on the target $M$ and Hypothesis $H$ in parallel and concurrently. Advanced DFS checks every time visitedState. If it could not be successful to find a counterexample after visiting all states in target $M$, then the last visited state is considered as an initial state, for function several times checking, if and only if it was not in the same partition with the current visited state in Hypothesis $H$ checked by MhatSTAT. Actually, function Several times checking continues the process of finding a counterexample, the same as Advanced DFS, if all states have been visited and the equivalence checker was not successful to find a counterexample.

<table>
<thead>
<tr>
<th>Target M:</th>
<th>Row</th>
<th>Input state</th>
<th>Input alphabet</th>
<th>Output state</th>
<th>Partition Check</th>
<th>A/R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q_1$</td>
<td>$a$</td>
<td>$q_2$</td>
<td>$q_2, a$</td>
<td>$R, R$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$q_2$</td>
<td>$a$</td>
<td>$q_3$</td>
<td>$q_3, aa$</td>
<td>$A, A$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$q_3$</td>
<td>$a$</td>
<td>$q_1$</td>
<td>$q_1, \lambda$</td>
<td>$R, R$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$q_1$</td>
<td>$b$</td>
<td>$q_4$</td>
<td>$q_4, \lambda$</td>
<td>$R, R$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$q_4$</td>
<td>$a$</td>
<td>$q_0$</td>
<td>$q_0, a$</td>
<td>$R, R$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$q_0$</td>
<td>$a$</td>
<td>$q_2$</td>
<td>$q_2, aa$</td>
<td>$A, R$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypothesis H:</th>
<th>Row</th>
<th>Input state</th>
<th>Input alphabet</th>
<th>Output state</th>
<th>Partition Check</th>
<th>A/R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\lambda$</td>
<td>$a$</td>
<td>$a$</td>
<td>$a, q_2$</td>
<td>$R, R$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$A$</td>
<td>$a$</td>
<td>$aa$</td>
<td>$aa, q_3$</td>
<td>$A, A$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$Aa$</td>
<td>$a$</td>
<td>$\lambda$</td>
<td>$\lambda, q_1$</td>
<td>$R, R$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$\lambda$</td>
<td>$b$</td>
<td>$\lambda$</td>
<td>$\lambda, q_4$</td>
<td>$R, R$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$\lambda$</td>
<td>$a$</td>
<td>$a$</td>
<td>$a, q_0$</td>
<td>$R, R$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$A$</td>
<td>$a$</td>
<td>$aa$</td>
<td>$aa, q_2$</td>
<td>$A, R$</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Partition check table for target $M$ and hypothesis $H$. $A = \text{Accept}$, $R = \text{Reject}$. 
Figure 4.3.a: Make an equivalence query on the target $M$.

Figure 4.3.b: Make an equivalence query on the hypothesis $H$. 
Chapter 5

5 Experimental Results

Note: All figures for the L* algorithm have been copied from [8] in order to compare with Kearns algorithm results.

The L* algorithm is typical of active learning. The goal of L* algorithm is to learn the unidentified systems which can be modelled as a deterministic finite automata (DFA) by asking queries from teacher and oracle, called membership and equivalence queries respectively. The L* algorithm has a simple structure and implementation to learn the unknown systems. However, we decide to evaluate and implement Kearns algorithm which has more complex, difficult structure and harder implementation than L* algorithm, more efficient. In this Chapter, we study the empirical behaviour of Kearns algorithm and analyse the experimental results of learning the random generated DFA $A$. Then we compare Kearns algorithm with L* algorithm with respect to difficulty of the learning considering membership and equivalence queries. The results of implementations show that Kearns algorithm is more efficient and faster than L* algorithm to learn a deterministic finite automata (DFA).

5.1 Experimental Method

Computational learning concerns modeling and designing algorithms to guess a target $M$, a particular structure $m \in M$, from a set of possible structures $M$ given some information $D = d_1 \ldots d_i \ldots d_n$ about $m$.

One of the simple methods to generate a target $M$ is manually generating samples which is too slow and takes a lot of time. This method is not so accurate because there is a high probability to have some errors during generation of the large samples manually and we generate information with errors or we miss some parts of samples. Since we need to test the large number of samples of target $M$ to get accurate average values, we use a random DFA generator to generate the target $M$ randomly in each execution. We consider computational learning algorithms that infer a deterministic finite automaton as a target $M$ and explain how a random DFA $A = (Q, \Sigma, \delta, q_0, F)$ is generated.

5.2 How to create random DFA?

A DFA is a finite automaton with:

- a certain set of states, $Q$.
- a finite set of input alphabets, $\Sigma$.
- a transition function (the input is a state and a transition and returns a state as an output; $\delta (q_0, a1) = q_1$).
- an initial state (one of the states like $q_0$).
• One or more final states (a set $F$ subset of $Q$).

So from a mathematical point of view, a Deterministic Finite Automata $A$ is shown as below:

$$A = (Q, \sum, \delta, q_0, F).$$

In the graphical user interface for a random DFA generator implemented in Java, according to available properties in $A$, there are three variables in order to insert the number of states $S$, input alphabets $R$, and final states $N$ for each location.

After running the programme:

The states of the DFA are arranged from $q_0$ to $q_{S-1}$, and then the alphabet is selected from alphabet $a$ up to $R^{th}$ alphabet character using the among English alphabet respectively. For example, for alphabet size $R = 4$, $\{a,b,c,d\}$ are considered.

The initial state is one of the generated states randomly chosen. The final states are selected based on the number of final states $N$ from defined states randomly. For example, if $N = 2$ then DFA is designed with two final states. Finally, a transition function is created for each state, starting from initial state, and all input alphabet symbols separately. The target state is one of the states of DFA randomly:

$$\delta (\text{state}, \text{alphabet}) = \text{next state randomly}.$$ 

Note: According to the definition of a DFA, for each state we should consider all input alphabet symbols which direct this state to other states.

### 5.3 Experimental Setup Using Random DFA Generation

In order to analyse the experimental results of learning a randomly generated DFA $A$ by Kearns algorithm, we decided to plot the number of membership and equivalence queries achieved by using different sizes of state set $|Q|$, alphabet size $|\sum|$ and the ratio of final states to all states, $r = |F|/|Q|$, to investigate the difficulty of learning. The difficulty of learning a DFA $A$ is determined by the number of membership and equivalence queries. Investigations of the number of membership queries from the experimental results of learning a randomly generated DFA $A = (Q, \sum, \delta, q_0, F)$ by Kearns learning algorithm prove that if Kearns algorithm starts to learn a DFA $A$ with one final state, whatever the number of final states is growing, it becomes easier for Kearns algorithm to learn $A$ until the number of final states is almost half of the number of all states. After that, $A$ becomes harder to learn by Kearns algorithm up to the hardest part of learning which is when the number of final states is almost equal to the number of all states.

Considering the number of required equivalence queries from the experimental results of learning a random generated DFA $A$, Kearns learning algorithm can discover a new state of a random DFA $A$ by having access to the equivalence oracle in order to answer an equivalence query on each iteration and at the end of
Chapter 5. Experimental Results

Execution, the number of equivalence queries is equal to the number of states. Therefore, there is no minimum and maximum required equivalence queries to measure the difficulty of learning and all samples of the same state size will be learned in the same difficulty.

5.4 Comparison of the L* and Kearns algorithms with respect to difficulty of the learning considering membership queries

We wished to compare the performance of Kearns algorithm with Angluin's L* DFA learning algorithm. This was because L* is simple and widely used.

In each experiment, the random DFA generator was used to generate a DFA $A = (Q, \sum, \delta, q_0, F)$ and learning algorithms were used to learn the generated automaton by having access to the teacher and oracle in order to answer membership and equivalence queries. The empirical behaviour of these learning algorithms depends on the size of state $|Q|$, alphabet size $|\sum|$ and the ratio of final states to all states; $r = |F|/|Q|$. These properties can have different effects on difficulty of the learning.

In the charts shown in Figure 5.1, an attempt has been made to demonstrate the comparison the L* and Kearns algorithms with respect to the number of membership queries achieved by using different state size $|Q|$, $0% < r < 100\%$, and fixed size of 10 alphabet. The results from implementations show that for both algorithms the number of membership queries gives an almost quadratic curve in chart and the minimum required queries in $r = 50\%$, called "easy to learn DFA", for 25, 50 and 75 state size. When the learner asks the minimum queries from teacher, it takes less computations and time, and therefore it is easier to learn. On the other hand, maximum required membership queries, called "hard to learn DFA", take place at $r = 99\%$ and $r = 1\%$ for Kearns and L* algorithms respectively. So, when the learner asks the maximum required queries from teacher, it takes more computations, time, and therefore it is harder to learn.

In each iteration, Kearns learning algorithm can discover a new state of the DFA $A$ (random DFA) by having access to the oracle in order to answer an equivalence query. Therefore, the number of times the Main Loop in Kearns algorithm is executed exactly corresponds to the number of states of $A$ (size $M$). Each execution of the Main Loop requires the processing of a counterexample by procedures Update-Tree $(\gamma, T)$ and Tentative Hypothesis $(T)$. A counterexample of length $n$ requires at most $n$ calls of Subroutine Sift$(s, T)$. As we have already explained, sifting operations use membership query in each execution of the Main Loop in Subroutine Sift$(s, T)$ (see section 4.1). In conclusion, the number of membership queries used in Kearns algorithm is directly related to the length of the counterexamples. As the number of states increases, the length of counterexample increases. Thus, the number of membership queries rises.
If $\gamma$ is a counterexample generated by equivalence checker in each execution of the Main Loop and the length of counterexamples is $LC$, when Kearns algorithm could learn the DFA $A$:

$$LC = |\sum_{i=1}^{n} \gamma_i|, n = n \text{ times execution of Main Loop to learn the DFA } A.$$ 

Kearns algorithm starts to learn the DFA $A$ with one final state. Whatever the number of final states is growing, it becomes easier for Kearns algorithm to learn $A$. It is because after learning the DFA $A$, the number of membership queries is decreased due to reduction of $LC$. The shortest $LC$ is when the number of final states is almost half of the number of all states, in $r = 50\%$. After that, the number of membership queries is increasing due to increment of $LC$. Therefore, $A$ becomes harder to learn by Kearns algorithm up to the hardest part of learning which is when the equivalence checker creates the longest $LC$, in $r = 99\%$.

The results from implementations show that L* algorithm has the same result: the number of membership queries gives an almost quadratic curve in chart. L* algorithm starts to learn the DFA $A$ with one final state, in $r = 1\%$. There is a high probability that, the DFA $A$ rejects an input alphabet in any state of a hypothetical acceptor and the DFA $A$ ends up in a rejecting state — i.e. the learning progresses only slowly or by counterexamples as the observation table is very likely consistent. Whatever the number of final states is growing, it becomes easier for L* algorithm to learn the DFA $A$. When the number of final states is almost half of the number of all states, in $r = 50\%$, the learner can investigate the different prefixes set faster with the consistency checks. There is the highest chance to find an inconsistency in $\mathcal{O}(T)$ at any time which drives the learner on without the need for many counterexamples, when $r = 50\%$. Despite the increasing the number of final states, for $r > 50\%$, the number of membership queries is increasing because again the learning progresses only slowly or by counterexamples as the observation table is very likely consistent. In conclusion, the number of membership queries used in L* algorithm depends on the number of counterexamples, new prefixes and suffixes.

According to the observations, the number of membership queries used in the Kearns algorithm are much less than in L* algorithm. Although the learning algorithms have almost the same behavior in learning DFA $A$, but considering the number of required membership queries achieved by different $r$ and sizes of state space $Q$ proves that Kearns algorithm is faster than L* algorithm.

The interesting part of comparison between the two algorithms in Figure 5.2 is when the algorithms are given “hard to learn DFA”. In this situation, when $r = 1\%$, it is hardest for L* algorithm to learn DFA because it uses the highest number of membership queries, while it occurs in Kearns algorithm when $r = 99\%$. 
Chapter 5. Experimental Results

a) Kearns algorithm

The number of alphabets is 10 for all samples.

b) $L^*$ algorithm

Figure 5.1: Behaviour of $L^*$ and Kearns algorithms respect to membership queries achieved by using different $r$.

Figure 5.2 illustrates the number of membership queries, in $r = 50\%$, in order to explain the data in more detail with different sizes of state space $Q$ and alphabets $\Sigma$. 
In both graphs, the number of membership queries grows nearly linearly and has a trend upwards corresponding to the increase in the number of states and alphabet size. According to the observations, the number of membership queries used in Kearns algorithm is much less than in L* algorithm. Although the learning algorithms have almost the same behavior in learning DFA $A$, but considering the number of required membership queries achieved by different sizes of state space $Q$ proves that Kearns algorithm is faster than L* algorithm, in $r = 50\%$. We see almost the same tendency with different alphabet sizes $|\Sigma|$. The number of membership queries used in the Kearns algorithm is much less than in L* algorithm. Therefore, according to the observations, Kearns algorithm is faster than L* algorithm, in $r = 50\%$ (Figure 5.3).

![Graph showing membership queries in each implementation with $r = 50\%$.](image1)

**a) Kearns algorithm**

![Graph showing membership queries for randomly generated DFAs, $r = 50\%$.](image2)

**b) L* algorithm**

Figure 5.2: Average membership queries to correctly learn DFAs of different state size with $r = 50\%$. 
Chapter 5. Experimental Results

a) Kearns algorithm

b) L* algorithm

Figure 5.3: Average membership queries to correctly learn DFAs.
5.5 Comparison of the L* and Kearns algorithms with respect to difficulty of the learning considering equivalence queries

One of the interesting and surprising results is the number of equivalence queries used in Kearns algorithm. It has completely different behaviour compared with L* algorithm. As explained before, Kearns learning algorithm can discover a new state of the target $M$ (random DFA $A$) by having access to the oracle in order to answer an equivalence query on each iteration. Therefore, at the end of execution, the number of equivalence queries is equal to the number of states. The actual results from experiments show that Kearns algorithm yields a linear chart for the required number of equivalence queries (Figure 5.4.a). Therefore, there is no minimum and maximum in Kearns plot in order to measure the difficulty of learning and all samples will be learned with the same difficulty. While, L* algorithm has an almost quadratic curve in the chart with the minimum required queries in $r = 50\%$ (Figure 5.4.b), the same as membership queries achieved from L* algorithm explained in the previous section.

Angluin formally proved in [5] that the number of equivalence queries is at most $|Q|$. L* algorithm should check that the observation table $OT$ is closed and consistent in each iteration. If $OT$ is not closed or consistent, the learner is not able to build a hypothesis $H_i$ from $OT$ and discover a new state of the DFA $A$. In this situation, L* algorithm tries to establish closed or consistent in observation table for the next iteration. The learner only performs an equivalence query if the observation table $OT$ is closed and consistent. Therefore, the run time for each performance of an equivalence query really depends on when the observation table is closed and consistent, this time is variable. In conclusion, although the number of equivalence queries in L* is less than in Kearns algorithm, we are not able to compare the learning time of L* and Kearns algorithms, in the practical evaluation, respect to the number of equivalence queries. Moreover, the essential measured parameter, that determines the learning time, is the number of membership queries.
Chapter 5. Experimental Results

Figure 5.4: Behaviour of L* and Kearns algorithms respect to equivalence queries achieved by using different r.

a) Kearns algorithm

b) L* algorithm
Implementing and Evaluating Automaton Learning Algorithms for a Software Testing Platform

42

a) Kearns algorithm

![Equivalence queries in each implementation with r = 50%](image1)

Figure 5.5: Average equivalence queries to correctly learn DFAs of different alphabet size with $r = 50\%$.

b) L* algorithm

![Equivalence queries per learning run r = 50%](image2)
If we compare two learning algorithms graphs shown in Figure 5.5, the number of equivalence queries correspond with the number of states in Kearns algorithm (Figure 5.5.a), whereas in L* algorithm the number of equivalence queries depends on the number of times that observation table is closed and consistent (Figure 5.5.b).

In both algorithms, the alphabet size does not affect to the number of equivalence queries. In L* algorithm, this is not logical because a higher number of input alphabets leads to make a more complex representation of the DFA $A$ and thus, more information to learn for the L* algorithm. This behaviour might be referred to the properties of the observation table $OT$. The learner needs to observe more behaviour. On the other hand, the learner only makes an equivalence query if the observation table is closed and consistent. While, in each iteration, Kearns learning algorithm can discover a new state of the DFA $A$ (random DFA) by having access to the oracle in order to answer an equivalence query. Each execution of the Main Loop requires the processing of a counterexample by procedure $\text{Update-Tree} (\gamma, T)$. The result of processing of the procedure $\text{Update-Tree} (\gamma, T)$ leads to discover a new state of the DFA $A$. The number of equivalence queries is independent of the alphabet size.

In Figure 5.6.a, the learner makes an equivalence query to find new state on each iteration. Since the number of equivalence queries corresponds with the number of states in each execution and it does not depend on alphabet size, there is just one sample for each different alphabet size (5, 10... 25) and there is a linear chart to show the plot. In Figure 5.6.b, the values almost overlap each other in each sample and, finally, the result is shown as curve. The number of equivalence queries grows almost logarithmically in proportion to $|Q|$. Angluin formally proved in [3] that there will be at most $|Q| - 1$ counterexamples generated by equivalence checker. Thus, considering the last equivalence query, the number of equivalence queries $\leq |Q|$. If we compare Figures 5.6 with 5.7, there are no changes in equivalence queries gained by Kearns algorithm because changing $r$ does not affect the graph, while for L* algorithm the behaviour has changed and growth of states corresponds to the scattering of equivalence queries in each sample. In Figure 5.5.b, the alphabet size does not affect the number of equivalence queries. However, as we already argued, this is not logical and we can prove our claim by referring to Figure 5.7.b. According to the observations, the number of equivalence queries is dependent on the alphabet size.
Implementing and Evaluating Automaton Learning Algorithms for a Software Testing Platform

a) Kearns algorithm

b) L* algorithm

Figure 5.6: Average equivalence queries to correctly learn DFAs of different state size.
Chapter 5. Experimental Results

Figure 5.7: Average equivalence queries to correctly learn DFAs of different state size $r = 99\%$ and 1\% for Kearns and $L^*$ respectively.

- **Kearns algorithm**

- **$L^*$ algorithm**
Chapter 6

6 Novelty of this Thesis

In this project we implement and evaluate advanced DFA learning algorithm called Kearns algorithm and compare it with Angluin’s L* algorithm, presented by Dana Angluin [5].

Active learning is a particular instance of supervised machine learning in which a learning algorithm is capable of querying the teacher to achieve the related outputs. While, in the passive model, a learning algorithm has definitely no control over the sample of labelled examples drawn and provides only random examples of the target concepts.

The L* and Kearns algorithms are typical of active learning. Lots of people have studied L* algorithm. This algorithm has been described by Chen [6], and Berg [7]. It has also been implemented and evaluated by some researchers and students (e.g., Czerny[8]), whereas Kearns algorithm for learning DFA with membership and equivalence queries (according to our knowledge and searching in the internet, scientific articles and learning library websites such as LearnLib [12]), has not been implemented yet.

The L* algorithm has a simple structure and implementation to learn the unknown systems. For this reason, we decided to evaluate and implement Kearns algorithm which has more complex, difficult structure and harder implementation than L* algorithm, however it is more efficient and faster.

Kearns algorithm is executed in iterative phases. At the beginning of the iteration, the initialized binary classification tree $T$ and hypothesis $H$ are considered as the initial values $T_0$ and $H_0$ respectively. During each iteration, the learner discovers a new state of hypothesis $H_i$ which is the last detected state of DFA $A$ by checking $T_i$. Then, the equivalence checker (learner) poses an equivalence query to ask the oracle. The question is whether the hypothesis $H$ is equal to the DFA $A$ or not? If the answer is negative then the oracle generates a counterexample. Finally, the tree will be updated and assumed as current $T_{i+1}$. This process will continue and when $M \equiv H_i$ so the learner has exactly learned the DFA $A$.

Considering the definition of two learning algorithms, the L* algorithm should check that the observation table $OT$ is closed and consistent in each iteration. If $OT$ is not closed or consistent, the learner is not able to build a hypothesis $H_i$ from $OT$ and discover a new state of DFA $A$. Therefore, L* algorithm tries to establish closed or consistent in observation table for the next iteration. On the other hand, during each iteration in Kearns algorithm, the learner discovers a new state of hypothesis $H_i$ which is the last detected state of DFA $A$ by checking $T_i$. 
In L* algorithm, the run time for each performance of an equivalence query really depends on when the observation table is closed and consistent, this time is variable. In conclusion, although the number of equivalence queries in L* is less than in Kearns algorithm, we are not able to compare the learning time of L* and Kearns algorithms, in the practical evaluation, respect to the number of equivalence queries. Moreover, the essential measured parameter, that determines the learning time, is the number of membership queries.

The other main novelty of this thesis is that we consider Kearns algorithm performance using the methodology for empirical behaviour analysis described in Czerny[8]. Both learning algorithms start to learn a DFA $A$ with one final state. Whatever the number of final states is growing, it becomes easier for learning algorithms to learn $A$ until the number of final states is almost half of the number of all states. After that, $A$ becomes hard to learn by learning algorithms up to the number of final states is almost equal to the number of all states. According to the observations, the number of membership queries used in Kearns algorithm is much less than in L* algorithm. Although the learning algorithms have almost the same behavior in learning DFA $A$, but considering the number of required membership queries proves that Kearns algorithm is faster than L* algorithm.

We used and also designed and built different methods and functions in the body of Kearns algorithm as follow:

- Advanced DFS (Depth First Search) [14].
- LCA (least Common Ancestor) [13].
- Random DFA Generator.
- Pre-Order traversal.
Chapter 7

7 Conclusion and Future Work

7.1 Conclusion

In this thesis we implemented and evaluated an existing algorithm for learning deterministic finite automata (DFA), known as Kearns algorithm. The aim of this thesis was to investigate how effective Kearns algorithm is from a software testing perspective. Since we needed to test a large number of samples of DFA to get accurate average values, we used a random DFA generator to generate the target $M$ randomly in each execution. Then, we analysed the experimental results of learning the random generated DFA. Finally, we compared Kearns algorithm with the $L^*$ algorithm with respect to difficulty of the learning considering membership and equivalence queries.

7.1.1 Comparison of Membership Queries

Investigations of the number of membership queries from the experimental results of learning a randomly generated DFA $A = (Q, \Sigma, \delta, q_0, F)$ by Kearns learning algorithm and compare it with $L^*$ algorithm prove that both learning algorithms start to learn a DFA $A$ with one final state, whatever the number of final states is growing (increase of the number of final states causes increase of the number of input strings which is accepted by $A$, therefore the learner requires the less number of membership queries), it becomes easier for learning algorithms to learn $A$ until the number of final states is almost half of the number of all states. After that, $A$ becomes harder to learn by learning algorithms up to the number of final states is almost equal to the number of all states, because it is harder for learner to find a counterexample.

The comparisons between $L^*$ and Kearns algorithms are based on the number of membership queries used in each execution. The number of membership queries achieved by using different sizes of state set $|Q|$, alphabet size $|\Sigma|$ and the ratio of final states to all states, $r = |F|/|Q|$, are used to analyse the difficulty of learning. The results from implementations show that the number of membership queries used in Kearns algorithm are much less than in $L^*$ algorithm. Thus, Kearns algorithm is faster than $L^*$ algorithm.

7.1.2 Comparison of Equivalence Queries

Considering the number of required equivalence queries from the experimental results of learning a random generated DFA $A$, Kearns learning algorithm can
discover a new state of a random DFA $A$ by having access to the equivalence oracle in order to answer an equivalence query on each iteration and at the end of execution, the number of equivalence queries is equal to the number of states. Therefore, there is no minimum and maximum required equivalence queries to measure the difficulty of learning and all samples of the same state size will be learned in the same difficulty. While, the $L^*$ algorithm should check that the observation table $OT$ is closed and consistent in each iteration. If $OT$ is not closed or consistent, the learner is not able to build a hypothesis $H_i$ from $OT$ and discover a new state of DFA $A$. Therefore, $L^*$ algorithm tries to establish closed or consistent in observation table for the next iteration.

In $L^*$ algorithm, the run time for each performance of an equivalence query really depends on when the observation table is closed and consistent, this time is variable. In conclusion, although the number of equivalence queries in $L^*$ is less than in Kearns algorithm, we are not able to compare the learning time of $L^*$ and Kearns algorithms, in the practical evaluation, respect to the number of equivalence queries. Moreover, the essential measured parameter, that determines the learning time, is the number of membership queries.

The number of equivalence queries is independent of the alphabet size. While, In $L^*$ algorithm, a higher number of input alphabets leads to make a more complex representation of the target $M$. Thus, there are more information to learn for the $L^*$ algorithm. This behaviour might be referred to the properties of the observation table $OT$. The learner needs to observe more behaviour. On the other hand, the learner only makes an equivalence query if the observation table is closed and consistent. In conclusion, the number of equivalence queries is dependent on the alphabet size.

### 7.2 Future Work

One of the best developments in the field of learning is optimizing and enhancing the performance of learning algorithms e.g., designing the learning algorithms to learn the models with more complex properties such as more inputs and variety of outputs instead of just accepting or rejecting inputs. Kearns algorithm can be improved and redesigned such that it could learn automata with several outputs chosen from set of $\beta = \{A, B, C \ldots X, Y, Z\}$. So, the black box receives a large number of inputs from set of $\alpha = \{a, b, c \ldots\}$ and returns one eligible output for each input from set of $\beta$. The improved Kearns algorithm could learn more complex models with various outputs.
Chapter 8

8 Appendix

Advanced DFS function:

```java
public static String advancedDFS (String initM, String initMhat, String CE, ArrayList <String> visitedState){
    boolean vis = visitedState.contains(initM);
    String CE_tmp = "";
    String last_CE = "";
    if(vis == false)
    {
        visitedState.add(initM);
        for (int i = 0; i < binary.transition.size() && AA == true; i++)
        {
            String A = binary.trTable.get(initM + binary.transition.get(i));
            CE_tmp = CE + binary.transition.get(i);
            boolean out1 = Binary.FinalStates.contains(A);
            String B = mhat.fmTable.get(initMhat + binary.transition.get(i))
            boolean out2 = finMhat.contains(B);
            visited1.add(initMhat);
            partition.put(initM, initMhat);
            if(out1 != out2 )
            {
                AA = false;
                break;
            }
        }
        CE_tmp = FDFS(A, B, CE_tmp, visitedState);
    }
    return CE_tmp;
}
else if (partition.containsKey(initM))
{
    String MhatSTAT = partition.get(initM);
    if (MhatSTAT != initMhat)
    {
        System.out.println("CE_part: "+ CE);
        ArrayList <String> visitedpart = new ArrayList<String> ();
        visitedpart.clear();
        last_CE = FinalSearch(initM, initMhat, CE, visitedpart);
        System.out.println("LAST_CE: "+last_CE);
        AAA = true;
    }
    return last_CE;
} else {
    return "";
}
```

```java
public static String severalTimesChecking (String initialM, String initialMhat, String CE, ArrayList <String> visited){
    boolean vis = visited.contains(initialM);
```
String NCE_tmp = "";

if(vis == false)
{
    visited.add(initialM);
    for (int j = 0; j < binary.transition.size() & & AAA == true; j++)
    {
        String C = binary.trTable.get(initialM + binary.transition.get(j));
        NCE_tmp = C + binary.transition.get(j);
        boolean out3 = Binary.FinalStates.contains(C);
        String D = mhat.fmTable.get(initialMhat + binary.transition.get(j));
        boolean out4 = finMhat.contains(D);
        if(out3 != out4)
        {
            AAA = false;
            AA = false;
            break;
        }else {
            NCE_tmp = severalTimesChecking (C, D, NCE_tmp, visited);
        }
    }
    return NCE_tmp;
}

AA = false;
return ""

Average of 15 executions to demonstrate behaviour of membership queries with different r in Kearns algorithms:

<table>
<thead>
<tr>
<th>r/Number of states</th>
<th>25</th>
<th>50</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10%</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>20%</td>
<td>5</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>30%</td>
<td>7</td>
<td>15</td>
<td>22</td>
</tr>
<tr>
<td>40%</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>50%</td>
<td>12</td>
<td>25</td>
<td>37</td>
</tr>
<tr>
<td>60%</td>
<td>15</td>
<td>30</td>
<td>45</td>
</tr>
<tr>
<td>70%</td>
<td>17</td>
<td>35</td>
<td>52</td>
</tr>
<tr>
<td>80%</td>
<td>20</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>90%</td>
<td>22</td>
<td>45</td>
<td>67</td>
</tr>
<tr>
<td>99%</td>
<td>24</td>
<td>49</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 8.1: r for 25, 50 and 75 state size.
<table>
<thead>
<tr>
<th>r/Number of states</th>
<th>25</th>
<th>50</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>726</td>
<td>2815</td>
<td>5578</td>
</tr>
<tr>
<td>1%</td>
<td>543</td>
<td>2114</td>
<td>6375</td>
</tr>
<tr>
<td>1%</td>
<td>835</td>
<td>2692</td>
<td>4509</td>
</tr>
<tr>
<td>1%</td>
<td>587</td>
<td>3192</td>
<td>6878</td>
</tr>
<tr>
<td>1%</td>
<td>616</td>
<td>2209</td>
<td>6408</td>
</tr>
<tr>
<td>1%</td>
<td>561</td>
<td>1948</td>
<td>5808</td>
</tr>
<tr>
<td>1%</td>
<td>634</td>
<td>2933</td>
<td>7356</td>
</tr>
<tr>
<td>1%</td>
<td>468</td>
<td>1817</td>
<td>6403</td>
</tr>
<tr>
<td>1%</td>
<td>643</td>
<td>2158</td>
<td>6882</td>
</tr>
<tr>
<td>1%</td>
<td>748</td>
<td>2670</td>
<td>4549</td>
</tr>
<tr>
<td>1%</td>
<td>493</td>
<td>2941</td>
<td>5763</td>
</tr>
<tr>
<td>1%</td>
<td>769</td>
<td>1874</td>
<td>6093</td>
</tr>
<tr>
<td>1%</td>
<td>563</td>
<td>2349</td>
<td>5193</td>
</tr>
<tr>
<td>1%</td>
<td>616</td>
<td>2041</td>
<td>4981</td>
</tr>
<tr>
<td>1%</td>
<td>466</td>
<td>1884</td>
<td>4001</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>617.8667</td>
<td>2375.8</td>
<td>5785.133</td>
</tr>
<tr>
<td>10%</td>
<td>500</td>
<td>1997</td>
<td>3667</td>
</tr>
<tr>
<td>10%</td>
<td>504</td>
<td>1885</td>
<td>3621</td>
</tr>
<tr>
<td>10%</td>
<td>439</td>
<td>1797</td>
<td>3728</td>
</tr>
<tr>
<td>10%</td>
<td>665</td>
<td>1594</td>
<td>3724</td>
</tr>
<tr>
<td>10%</td>
<td>491</td>
<td>1575</td>
<td>3789</td>
</tr>
<tr>
<td>10%</td>
<td>611</td>
<td>2181</td>
<td>3556</td>
</tr>
<tr>
<td>10%</td>
<td>478</td>
<td>1551</td>
<td>3468</td>
</tr>
<tr>
<td>10%</td>
<td>531</td>
<td>1646</td>
<td>3568</td>
</tr>
<tr>
<td>10%</td>
<td>532</td>
<td>1616</td>
<td>3426</td>
</tr>
<tr>
<td>10%</td>
<td>535</td>
<td>1688</td>
<td>3729</td>
</tr>
<tr>
<td>10%</td>
<td>499</td>
<td>1965</td>
<td>3868</td>
</tr>
<tr>
<td>10%</td>
<td>559</td>
<td>2076</td>
<td>3841</td>
</tr>
<tr>
<td>10%</td>
<td>607</td>
<td>1583</td>
<td>3685</td>
</tr>
<tr>
<td>10%</td>
<td>471</td>
<td>1666</td>
<td>3680</td>
</tr>
<tr>
<td>10%</td>
<td>633</td>
<td>1705</td>
<td>3361</td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td>537</td>
<td>1768.333</td>
<td>3647.4</td>
</tr>
<tr>
<td>20%</td>
<td>399</td>
<td>1594</td>
<td>3228</td>
</tr>
<tr>
<td>20%</td>
<td>408</td>
<td>1526</td>
<td>3172</td>
</tr>
<tr>
<td>20%</td>
<td>472</td>
<td>1544</td>
<td>3567</td>
</tr>
<tr>
<td>20%</td>
<td>398</td>
<td>1610</td>
<td>3241</td>
</tr>
<tr>
<td>20%</td>
<td>387</td>
<td>1525</td>
<td>3147</td>
</tr>
<tr>
<td>20%</td>
<td>403</td>
<td>1463</td>
<td>3204</td>
</tr>
<tr>
<td>20%</td>
<td>436</td>
<td>1611</td>
<td>3339</td>
</tr>
<tr>
<td>20%</td>
<td>433</td>
<td>1440</td>
<td>3284</td>
</tr>
<tr>
<td>20%</td>
<td>432</td>
<td>1495</td>
<td>3093</td>
</tr>
<tr>
<td>20%</td>
<td>471</td>
<td>1766</td>
<td>3452</td>
</tr>
<tr>
<td>20%</td>
<td>416</td>
<td>1671</td>
<td>3272</td>
</tr>
<tr>
<td>Percentage</td>
<td>Number 1</td>
<td>Number 2</td>
<td>Number 3</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>20%</td>
<td>438</td>
<td>1529</td>
<td>3185</td>
</tr>
<tr>
<td>20%</td>
<td>454</td>
<td>1642</td>
<td>3417</td>
</tr>
<tr>
<td>20%</td>
<td>462</td>
<td>1417</td>
<td>3340</td>
</tr>
<tr>
<td>20%</td>
<td>428</td>
<td>1535</td>
<td>3211</td>
</tr>
<tr>
<td>Average:</td>
<td>429.1333</td>
<td>1557.867</td>
<td>3276.8</td>
</tr>
<tr>
<td>30%</td>
<td>410</td>
<td>1455</td>
<td>3348</td>
</tr>
<tr>
<td>30%</td>
<td>354</td>
<td>1512</td>
<td>3266</td>
</tr>
<tr>
<td>30%</td>
<td>362</td>
<td>1437</td>
<td>3305</td>
</tr>
<tr>
<td>30%</td>
<td>396</td>
<td>1530</td>
<td>3228</td>
</tr>
<tr>
<td>30%</td>
<td>374</td>
<td>1480</td>
<td>3210</td>
</tr>
<tr>
<td>30%</td>
<td>370</td>
<td>1499</td>
<td>3012</td>
</tr>
<tr>
<td>30%</td>
<td>446</td>
<td>1461</td>
<td>3208</td>
</tr>
<tr>
<td>30%</td>
<td>429</td>
<td>1565</td>
<td>3259</td>
</tr>
<tr>
<td>30%</td>
<td>445</td>
<td>1539</td>
<td>3157</td>
</tr>
<tr>
<td>30%</td>
<td>404</td>
<td>1456</td>
<td>3126</td>
</tr>
<tr>
<td>30%</td>
<td>497</td>
<td>1496</td>
<td>3100</td>
</tr>
<tr>
<td>30%</td>
<td>409</td>
<td>1507</td>
<td>2977</td>
</tr>
<tr>
<td>30%</td>
<td>399</td>
<td>1451</td>
<td>3268</td>
</tr>
<tr>
<td>30%</td>
<td>414</td>
<td>1542</td>
<td>3336</td>
</tr>
<tr>
<td>30%</td>
<td>391</td>
<td>1414</td>
<td>3247</td>
</tr>
<tr>
<td>Average:</td>
<td>406.6667</td>
<td>1489.6</td>
<td>3203.133</td>
</tr>
<tr>
<td>40%</td>
<td>402</td>
<td>1474</td>
<td>3208</td>
</tr>
<tr>
<td>40%</td>
<td>398</td>
<td>1453</td>
<td>3066</td>
</tr>
<tr>
<td>40%</td>
<td>405</td>
<td>1488</td>
<td>3276</td>
</tr>
<tr>
<td>40%</td>
<td>389</td>
<td>1473</td>
<td>3152</td>
</tr>
<tr>
<td>40%</td>
<td>385</td>
<td>1668</td>
<td>3071</td>
</tr>
<tr>
<td>40%</td>
<td>388</td>
<td>1375</td>
<td>3324</td>
</tr>
<tr>
<td>40%</td>
<td>411</td>
<td>1468</td>
<td>3226</td>
</tr>
<tr>
<td>40%</td>
<td>390</td>
<td>1420</td>
<td>3260</td>
</tr>
<tr>
<td>40%</td>
<td>378</td>
<td>1451</td>
<td>3345</td>
</tr>
<tr>
<td>40%</td>
<td>466</td>
<td>1492</td>
<td>3100</td>
</tr>
<tr>
<td>40%</td>
<td>370</td>
<td>1560</td>
<td>3063</td>
</tr>
<tr>
<td>40%</td>
<td>383</td>
<td>1503</td>
<td>3101</td>
</tr>
<tr>
<td>40%</td>
<td>432</td>
<td>1452</td>
<td>3209</td>
</tr>
<tr>
<td>40%</td>
<td>381</td>
<td>1450</td>
<td>3191</td>
</tr>
<tr>
<td>40%</td>
<td>364</td>
<td>1346</td>
<td>3234</td>
</tr>
<tr>
<td>Average:</td>
<td>396.1333</td>
<td>1471.533</td>
<td>3188.4</td>
</tr>
<tr>
<td>50%</td>
<td>383</td>
<td>1535</td>
<td>3111</td>
</tr>
<tr>
<td>50%</td>
<td>397</td>
<td>1516</td>
<td>3120</td>
</tr>
<tr>
<td>50%</td>
<td>368</td>
<td>1405</td>
<td>3044</td>
</tr>
<tr>
<td>50%</td>
<td>360</td>
<td>1416</td>
<td>3260</td>
</tr>
<tr>
<td>50%</td>
<td>381</td>
<td>1448</td>
<td>3031</td>
</tr>
<tr>
<td>50%</td>
<td>362</td>
<td>1435</td>
<td>2972</td>
</tr>
<tr>
<td>50%</td>
<td>365</td>
<td>1431</td>
<td>3046</td>
</tr>
<tr>
<td>50%</td>
<td>370</td>
<td>1413</td>
<td>3009</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>60%</td>
<td>70%</td>
</tr>
<tr>
<td>----</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td>402</td>
<td>375</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>1425</td>
<td>1503</td>
<td>1427</td>
</tr>
<tr>
<td></td>
<td>3024</td>
<td>3247</td>
<td>3397</td>
</tr>
<tr>
<td></td>
<td>1436</td>
<td>1424</td>
<td>1436</td>
</tr>
<tr>
<td></td>
<td>3103</td>
<td>3108</td>
<td>3209</td>
</tr>
<tr>
<td></td>
<td>1488</td>
<td>1591</td>
<td>1485</td>
</tr>
<tr>
<td></td>
<td>3069</td>
<td>2988</td>
<td>3097</td>
</tr>
<tr>
<td></td>
<td>1442</td>
<td>1478</td>
<td>1576</td>
</tr>
<tr>
<td></td>
<td>3109</td>
<td>3309</td>
<td>3126</td>
</tr>
<tr>
<td></td>
<td>1432</td>
<td>1503</td>
<td>1573</td>
</tr>
<tr>
<td></td>
<td>3075</td>
<td>3069</td>
<td>3076</td>
</tr>
<tr>
<td></td>
<td>1434</td>
<td>1503</td>
<td>1416</td>
</tr>
<tr>
<td></td>
<td>3039</td>
<td>3115</td>
<td>3215</td>
</tr>
<tr>
<td></td>
<td>1451</td>
<td>1451</td>
<td>1512</td>
</tr>
<tr>
<td></td>
<td>3130</td>
<td>3130</td>
<td>3076</td>
</tr>
<tr>
<td></td>
<td>1495</td>
<td>1572</td>
<td>1515</td>
</tr>
<tr>
<td></td>
<td>3173</td>
<td>3265</td>
<td>3062</td>
</tr>
<tr>
<td></td>
<td>1449</td>
<td>1499</td>
<td>1575</td>
</tr>
<tr>
<td></td>
<td>3097</td>
<td>3115</td>
<td>3045</td>
</tr>
<tr>
<td></td>
<td>1434</td>
<td>1576</td>
<td>1369</td>
</tr>
<tr>
<td></td>
<td>3039</td>
<td>3126</td>
<td>3353</td>
</tr>
</tbody>
</table>

Average: 387.5333 1450.733 3092.6

<table>
<thead>
<tr>
<th></th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>403</td>
<td>421</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>1495</td>
<td>1573</td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td>3209</td>
<td>3215</td>
<td>3050</td>
</tr>
<tr>
<td></td>
<td>1436</td>
<td>1512</td>
<td>1618</td>
</tr>
<tr>
<td></td>
<td>3266</td>
<td>3321</td>
<td>3313</td>
</tr>
<tr>
<td></td>
<td>1432</td>
<td>1529</td>
<td>1512</td>
</tr>
<tr>
<td></td>
<td>3109</td>
<td>3076</td>
<td>3321</td>
</tr>
<tr>
<td></td>
<td>1449</td>
<td>1575</td>
<td>1486</td>
</tr>
<tr>
<td></td>
<td>3115</td>
<td>3045</td>
<td>3278</td>
</tr>
<tr>
<td></td>
<td>1434</td>
<td>1369</td>
<td>1479</td>
</tr>
<tr>
<td></td>
<td>3039</td>
<td>3353</td>
<td>3525</td>
</tr>
<tr>
<td></td>
<td>1495</td>
<td>1479</td>
<td>3313</td>
</tr>
<tr>
<td></td>
<td>3097</td>
<td>3525</td>
<td>3321</td>
</tr>
</tbody>
</table>

Average: 394.2667 1480.133 3132.267

<table>
<thead>
<tr>
<th></th>
<th>70%</th>
<th>80%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>420</td>
<td>385</td>
</tr>
<tr>
<td></td>
<td>1429</td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td>3063</td>
<td>3050</td>
</tr>
<tr>
<td></td>
<td>1428</td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td>3265</td>
<td>3215</td>
</tr>
<tr>
<td></td>
<td>1427</td>
<td>1575</td>
</tr>
<tr>
<td></td>
<td>3397</td>
<td>3045</td>
</tr>
<tr>
<td></td>
<td>1436</td>
<td>1369</td>
</tr>
<tr>
<td></td>
<td>3266</td>
<td>3353</td>
</tr>
<tr>
<td></td>
<td>1486</td>
<td>1479</td>
</tr>
<tr>
<td></td>
<td>3278</td>
<td>3525</td>
</tr>
</tbody>
</table>

Average: 403.6667 1474.2 3177.4
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>80%</td>
<td>455</td>
<td>1525</td>
<td>3141</td>
</tr>
<tr>
<td>80%</td>
<td>387</td>
<td>1448</td>
<td>3144</td>
</tr>
<tr>
<td>80%</td>
<td>421</td>
<td>1553</td>
<td>3277</td>
</tr>
<tr>
<td>80%</td>
<td>476</td>
<td>1546</td>
<td>3247</td>
</tr>
<tr>
<td>80%</td>
<td>450</td>
<td>1507</td>
<td>3564</td>
</tr>
<tr>
<td>80%</td>
<td>444</td>
<td>1520</td>
<td>3080</td>
</tr>
<tr>
<td>80%</td>
<td>443</td>
<td>1509</td>
<td>3347</td>
</tr>
<tr>
<td>80%</td>
<td>435</td>
<td>1542</td>
<td>3114</td>
</tr>
<tr>
<td>80%</td>
<td>391</td>
<td>1482</td>
<td>3594</td>
</tr>
<tr>
<td>80%</td>
<td>405</td>
<td>1389</td>
<td>3373</td>
</tr>
<tr>
<td>Average:</td>
<td>428.3333</td>
<td>1501.067</td>
<td>3304.533</td>
</tr>
<tr>
<td>90%</td>
<td>496</td>
<td>1534</td>
<td>3634</td>
</tr>
<tr>
<td>90%</td>
<td>438</td>
<td>1714</td>
<td>3360</td>
</tr>
<tr>
<td>90%</td>
<td>453</td>
<td>1875</td>
<td>3785</td>
</tr>
<tr>
<td>90%</td>
<td>548</td>
<td>1606</td>
<td>3390</td>
</tr>
<tr>
<td>90%</td>
<td>410</td>
<td>1672</td>
<td>3225</td>
</tr>
<tr>
<td>90%</td>
<td>481</td>
<td>1735</td>
<td>3804</td>
</tr>
<tr>
<td>90%</td>
<td>477</td>
<td>1848</td>
<td>3508</td>
</tr>
<tr>
<td>90%</td>
<td>439</td>
<td>1724</td>
<td>3354</td>
</tr>
<tr>
<td>90%</td>
<td>493</td>
<td>1772</td>
<td>3500</td>
</tr>
<tr>
<td>90%</td>
<td>413</td>
<td>2126</td>
<td>3708</td>
</tr>
<tr>
<td>90%</td>
<td>422</td>
<td>1534</td>
<td>3238</td>
</tr>
<tr>
<td>90%</td>
<td>412</td>
<td>1709</td>
<td>3694</td>
</tr>
<tr>
<td>90%</td>
<td>459</td>
<td>1917</td>
<td>3292</td>
</tr>
<tr>
<td>90%</td>
<td>556</td>
<td>1672</td>
<td>3657</td>
</tr>
<tr>
<td>90%</td>
<td>469</td>
<td>1628</td>
<td>3694</td>
</tr>
<tr>
<td>Average:</td>
<td>464.4</td>
<td>1737.733</td>
<td>3522.867</td>
</tr>
<tr>
<td>99%</td>
<td>462</td>
<td>2538</td>
<td>7009</td>
</tr>
<tr>
<td>99%</td>
<td>513</td>
<td>2160</td>
<td>6368</td>
</tr>
<tr>
<td>99%</td>
<td>572</td>
<td>3178</td>
<td>5437</td>
</tr>
<tr>
<td>99%</td>
<td>771</td>
<td>2692</td>
<td>5613</td>
</tr>
<tr>
<td>99%</td>
<td>622</td>
<td>2612</td>
<td>5251</td>
</tr>
<tr>
<td>99%</td>
<td>797</td>
<td>2011</td>
<td>6663</td>
</tr>
<tr>
<td>99%</td>
<td>640</td>
<td>2735</td>
<td>6104</td>
</tr>
<tr>
<td>99%</td>
<td>480</td>
<td>1971</td>
<td>6774</td>
</tr>
<tr>
<td>99%</td>
<td>409</td>
<td>2130</td>
<td>6211</td>
</tr>
<tr>
<td>99%</td>
<td>630</td>
<td>3121</td>
<td>5504</td>
</tr>
<tr>
<td>99%</td>
<td>688</td>
<td>2315</td>
<td>5424</td>
</tr>
<tr>
<td>99%</td>
<td>696</td>
<td>2974</td>
<td>4914</td>
</tr>
<tr>
<td>99%</td>
<td>745</td>
<td>1896</td>
<td>7046</td>
</tr>
<tr>
<td>99%</td>
<td>670</td>
<td>1644</td>
<td>5454</td>
</tr>
<tr>
<td>99%</td>
<td>465</td>
<td>1816</td>
<td>5125</td>
</tr>
<tr>
<td>Average:</td>
<td>610.6667</td>
<td>2386.2</td>
<td>5928.267</td>
</tr>
</tbody>
</table>

Table 8.2: Average of 15 executions for 25, 50 and 75 state size.
Table 8.3: Number of membership queries with different $r$ in Kearns algorithms.

<table>
<thead>
<tr>
<th>$r$/Number of states</th>
<th>25</th>
<th>50</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>617.8667</td>
<td>2375.8</td>
<td>5785.133</td>
</tr>
<tr>
<td>10%</td>
<td>537</td>
<td>1768.333</td>
<td>3647.4</td>
</tr>
<tr>
<td>20%</td>
<td>429.1333</td>
<td>1557.867</td>
<td>3276.8</td>
</tr>
<tr>
<td>30%</td>
<td>406.6667</td>
<td>1489.6</td>
<td>3203.133</td>
</tr>
<tr>
<td>40%</td>
<td>396.1333</td>
<td>1471.533</td>
<td>3188.4</td>
</tr>
<tr>
<td>50%</td>
<td>387.5333</td>
<td>1450.733</td>
<td>3092.6</td>
</tr>
<tr>
<td>60%</td>
<td>394.2667</td>
<td>1480.133</td>
<td>3132.267</td>
</tr>
<tr>
<td>70%</td>
<td>403.6667</td>
<td>1474.2</td>
<td>3177.4</td>
</tr>
<tr>
<td>80%</td>
<td>428.3333</td>
<td>1501.067</td>
<td>3304.533</td>
</tr>
<tr>
<td>90%</td>
<td>464.4</td>
<td>1737.733</td>
<td>3522.867</td>
</tr>
<tr>
<td>99%</td>
<td>610.6667</td>
<td>2386.2</td>
<td>5928.267</td>
</tr>
</tbody>
</table>

The number of alphabets is 10 for all samples.

Behaviour of membership queries with different $r$ in Kearns algorithms.
9 Bibliography


# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBT</td>
<td>Learning Base Test</td>
</tr>
<tr>
<td>DFA</td>
<td>Deterministic Finite Automata</td>
</tr>
<tr>
<td>RBT</td>
<td>Requirements-based Testing</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>LCA</td>
<td>Least Common Ancestor</td>
</tr>
<tr>
<td>EC</td>
<td>Equivalence Checker</td>
</tr>
<tr>
<td>LC</td>
<td>Length of counterexamples</td>
</tr>
<tr>
<td>MQ</td>
<td>Membership Query</td>
</tr>
</tbody>
</table>
My interesting courses in my studies were Learning Systems (Machine Learning) and Theory of Language and Machines. Therefore, I decided to choose my final thesis in the field of Automaton Learning Algorithms which was related to my favourite subjects.