BACHELOR'S THESIS COMPUTING SCIENCE

Tackling the Far Output Distinction Problem

Author:
Tomas Woldu
s1024181

Supervisor:
prof. dr. Frits Vaandrager
f.vaandrager@cs.ru.nl

Second reader:
dr. Jurriaan Rot
j.rot@cs.ru.nl

June 11, 2023

Radboud University

Abstract

Active automata learning is a technique used to learn models using membership/equivalence queries. Active learning consists of two phases, namely the exploration phase and testing phase. In the exploration phase, the learner constructs a hypothesis of the SUL (system under learning) by posing membership queries to the SUL. In the testing phase, the correctness of the constructed hypothesis is tested by posing equivalence queries and a counter-example is produced if a di erence is observed, otherwise the learning is concluded. When learning models that exhibit far output distinction behavior, nding counter-examples between the constructed hypothesis and the SUL requires exploring many long test sequences, of which only one may re ne the hypothesis further. In this thesis, we explore the impact that the inclusion of the most occurring subsequences in the test sequence may have in the learning of such models. We extract the most occurring subsequences from a generalized su x tree built from the logs/traces of such models. We test our approach by running experiments on a family of Mealy machine that exhibit the far output distinction behavior. We conclude that our approach indeed helps with the learning of models that exhibit the far output distinction behavior.

Contents

Chapter 1

Introduction

Specifying behaviors of large scale software components such as ASML's TWINSCAN machine is complex due to these components often consisting of many other interacting components under the hood. However, representing these components using a Mealy machine and applying the concept of the active automata learning can help us derive behavioral specifications of these components.

Active automata learning can be traced back to Angulin's seminal paper[2] on learning regular set through queries and counterexamples. Since then, the concept of active learning has been successfully applied in many fields across computer science. Common problems in computer science such as finding bugs in software[1], finding security vulnerabilities in network protocol[4] have been explored using active automata learning. The concept of active automata learning was also applied in evaluating whether legacy software and refactored implementation behave the same [6].

While active automata learning have been applied successfully in some of the aforementioned problems, there are also instances where active automata learning was not successful, for instance the learning of some of the industrial cyber-physical components in Aslam's thesis [3]. Although the application of automata learning is quite effective, there are some components that were not learned using state-of-the-art learning and testing algorithms. A key problem, as pointed out by Aslam, is what is called the *far output distinction behavior* exhibited by some of the components. In this thesis, we are attempting to solve this very problem. The components as well as the hypothesis models are represented by Mealy machines. From this section onwards, we refer to components as models. The following sections will introduce the problem and guide you through the hypothesis of the proposed solution we have experimented with in this thesis.

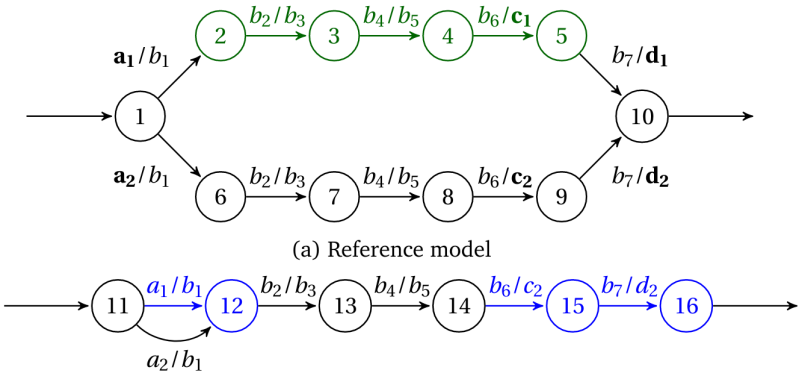

Figure 1.1: Far-output-distinction example [3]

1.1 Far output distinction behavior

Models where earlier choices a ect the behavior much later on are said to show the far output distinction behavior. For completeness and ease of understanding, we point to Figure 1.1 provided in Aslam's thesis, which displays a structural di erence in the learned and reference model for one of the component that exhibit the far-output distinction behavior (FODB). Figure 1.1 (a) is the reference model and Figure 1.1 (b) is the learned model. As we can see from the gure, the path coloured green is missed by the algorithm, because a choice made in state 1 has an e ect much later on. Hence, the algorithm fails to explore the path $a_1b_2b_4b_6$, that distinguishes state 2 from state 6.

The active automata learning cycle consists of the following steps:

1. The model learning algorithm (learner) constructs a hypothesis model of the unknown component, referred to as system under learning (SUL).

2. Conformance testing techniques (CTT) are then used to construct test suites, which are used to check for equivalence between the hypothesis model and the SUL. If a counterexample is found, the hypothesis model is then re ned and the learning cycle continues. Otherwise, the learning ends.

A test suite T is usually constructed as a product of three sets namely an access sequence set A, permutation of input symbols I^n up to length n and characterization set C for a n complete test suite, where n is the number of extra states a hypothesis may have. A test suite T is a n complete test suite if the hypothesis has at most n extra states with respect to SUL and the hypothesis passes T. A hypothesis passes T if there is no di erence in output observed between the hypothesis and SUL. A counterexample for

models with FODB consists of a long sequence of inputs capable of getting the system into states where di erent outputs can be seen. Finding these counterexamples has proven to be an expensive operation as, for a model with x input symbols, there are x^m combinations of sequences of length m. In the worst case, of all the possible combinations, only one serves as a counter example that can further re ne the hypothesis model [3]. The search space for these input sequences explodes exponentially with the increase of m.

1.2 Our solution to the problem

The key problem lies in the use of conformance testing algorithms, which try randomly generated input sequences in e orts to nd a counter-example. Our approach would be to extend this notion by including the subsequences (SS) of inputs that occur frequently from the logs of these models and use these subsequences as in xes during test generation. The test suite T then becomes a product of A, I^n, S and C, where S is a set of the most occurring subsequences of at least length k. A test $t = w{:}x{:}y{:}z \in T$ (. here represents concatenation) will then have $w \in A$, $x \in I^n$, $y \in S$ & $z \in C$. We can easily obtain these frequently occurring subsequences by:

- Constructing a generalized su x tree (GST) from the inputs that occur in the logs/traces of these models.
- Extracting a set of frequently occurring subsequence from the generalized su x tree.

Figure 1.2 displays the schematic view of our solution to the problem. The blue in the schema denotes the solution discussed above, whereas the **black** denotes the learning oracle. The research question to be answered in this thesis is as follows: ***How does the addition of frequently occurring subsequences in to the test sequences impact the learning of models that exhibit the far-output distinction behavior?***

Chapter 3 and Chapter 4 will cover the algorithms used to generate a generalized su x tree and extract subsequences from the GST. Chapter 5 will introduce and motivate the choice to conduct our experiments with the code-words family of Mealy machines. Succeeding this chapter comes Chapter 6 where we will discuss the setup of the experiments and discuss results of our experiment. Lastly, we end the document with Chapter 7 that discusses the conclusion from our experiments.

All the code related to generalized su x tree can be found in the following repository `https://github.com/Tomas2496/gsuffix`. All the code regarding the experiments can be found in `https://gitlab.science.ru.nl/bharat/tomas_bsc_thesis`.

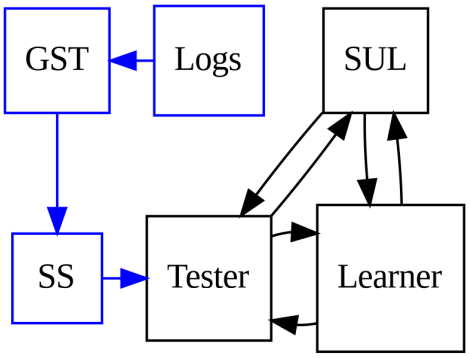

Figure 1.2: A schematic view of the solution

Chapter 2

Background

2.1 Mealy machines

We use the de nition of Mealy machine from [7]. A Mealy machine M is de ned as a 6-tuple $hS; s_0; ; ; ; i$ where,

- S is a nite nonempty set of states.
- $s_0 \ 2 \ S$ is the initial state.
- is a nite input alphabet.
- is a nite output alphabet.
- $: S \quad ! \ S$ is the transition function.
- $: S \quad !$ is the output function.

We extend to $: S \quad ! \ S$ to handle sequences of input symbols. Similarly, we do the same with , hence $: S \quad !$, where is the set of sequence of output symbols.

2.2 Access sequence set

For a Mealy machine $M = hS; s_0; ; ; ; i$, an access sequence for a state $s \ 2 \ S$ is a sequence of input symbols $i \ 2$ such that $(s_0; i) = s$. An access sequence set A is a set that contains an access sequence for every state $s \ 2 \ S$ in M.

2.3 Characterization set

For a Mealy machine $M = hS; s_0; ; ; ; i$, $i \ 2$ is a separating input sequence for states $s; s^0 \ 2 \ S; s \ 6= \ s^0$ if $(s; i) \ 6= \ (s^0; i)$. A characterization set C is a set of separating input sequence for every $s; s^0 \ 2 \ S$ and $s \ 6= \ s^0$.

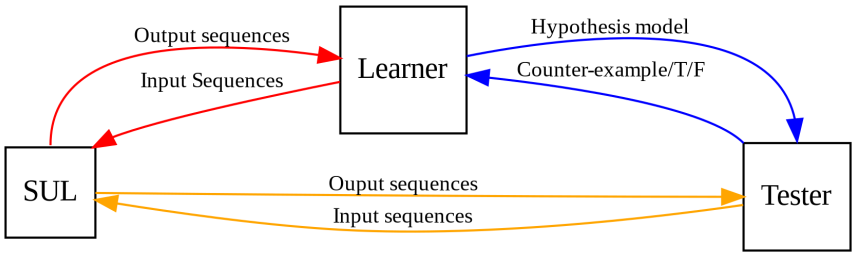

Figure 2.1: The active learning cycle

2.4 Active automata learning

Active learning is a black box technique used to learn models using query-response mechanism [3]. The queries are then responded to appropriately and correctly by oracles that ful ll the minimally adequate teacher (MAT) model [2, 7]. Active automata learning comprises the learning algorithm (learner) and the model to be learned, commonly referred to as system under learning (SUL). There are two kinds of queries :

1. **Membership queries** are utilized to retrieve behavioral information on the SUL. The learner queries the oracle with input sequences and the oracle responds with the corresponding output sequences. These output sequences are collected and analyzed to construct a hypothesis of the SUL [7].

2. **Equivalence queries** are used to check for equivalence between the hypothesis and the SUL. A counter example is returned if there are diverging behaviors between the hypothesis and SUL. The counter example is further used to re ne the hypothesis [7]. Otherwise, a boolean is returned to signify that the hypothesis and SUL are equivalent.

Figure 2.1 shows a schematic representation of the active learning cycle. The red edges represent the membership queries. The blue edges represent equivalence queries. The orange edges represent part of the equivalence queries the tester makes to check for equivalence. The tester has full access to the hypothesis model and can query input sequences to it. The tester queries the SUL with input sequence i that is part of a test suite that is constructed for equivalence checking between the hypothesis and the SUL. The SUL responds with the output sequence o for the queried input sequence i. The tester then compares o with the output sequence retrieved when the hypothesis model is fed with i. Based on the comparison, the tester then responds to the learner with a counter example or with a boolean signifying that the hypothesis and SUL are equivalent.

Chapter 3

Suffix Tree

The chapter introduces the su x tree data-structure and Ukkonen's algorithm that is one of the most popular algorithms used to construct a su x tree for a given word *!*. Furthermore, this chapter introduces terminologies, symbols and de nitions that are vital in understanding the upcoming chapters.

3.1 Definitions

Definition 2.1.1. A tree is a directed graph *G = (V, E)*, such that *V* contains a vertex , such that for each *v 2 V* there is a unique path from to *v*.

Definition 2.1.2. A *k-ary tree* is a tree where all nodes have an upper bound on the number of child nodes, namely *k*. When the edge labels contain more than one symbol, the k-ary tree is called compressed, for instance, the edge labels of a su x tree over word *!* are subsequences of *!*.

A *su x tree* is a compressed k-ary tree for word *!* where *k = j j*, where is the alphabet of word *!*. A su x tree stores all the su xes of a given string over the alphabet . In this thesis, we are going to use an altered version of the de nition provided by Ukkonen [8] for su x-tree. In Ukkonen's [8] de nition, he introduces the notion of an auxiliary state *?*, that we do not utilize in this thesis for simplicity purposes. Throughout this document, we will use $STree(c_i)$ to denote su x tree for some word *!* until the character c_i.

Definition 2.1.3. A su x-tree can be de ned using a 5-tuple *hQ; root; F; g; f i*, where

- *Q* is a set of (explicit/branching) states of the su x-tree.
- *root* is a state representing the root of the tree.
- *F* is a set of leaf states of the tree, *F Q*.

- *g* is a transition function that takes a reference pair of a state as input and transitions to the appropriate state in the tree.
- *f* is a function that of type *Q ! Q* that gives the su x link of the given state.

The numbers in the leaf state corresponds to the start index of the su x, for instance in g 3.2 the leaf state with index 2 represents the su x *nse$* of word *sense$*.

There are two types of states in a su x tree :

- Branching states (Inner nodes and root) and leaf states are explicit states.
- Implicit states are present in the edge labels. Consider the edge with label *nse$* in g:3.2. The edges in a su x tree are compressed, therefore there are hidden states in edges. These states can be seen in g:3.1.

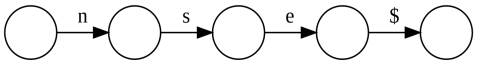

Figure 3.1: Implicit states of edge nse$

We are going to follow Ukkonen's way [8] of referencing states, as it goes very well with understanding the algorithm. We use the reference pair (s, w) to refer to an explicit or implicit state r of a su x tree, where s denotes an explicit state that is an ancestor of r and w denotes the string that is written out by the transitions from s to r in the appropriate su x tree. When s is r's nearest ancestor (making w the shortest feasible), a reference pair is said to be canonical. The canonical reference pair for an explicit r is, of course, (r,). Once more, we encode the string w as a pair of pointers (k, p) so that $t_k:::t_p = !$. A reference pair (s, w) acquires the form (s, (k, p)) in this manner. The symbol for the pair (s,) is (s, (p, p)).

A su x tree for a set of strings over the alphabet is called *a generalized su x tree*. The leaf state of a generalized su x tree stores a pair, (*wordNumber; startIndex*) as it can be seen in gure 3.3.

Definition 2.2.3. Let state *a* and *b* be two states such that the path to *a* is *x* and the path to state *b* is , where *jxj* = 1 and *j j* >= 0 then there is an edge from state *a* to *b* and this edge is called *su x link*. Su x links are represented by dotted arrows, as can be seen in gure 3.2.

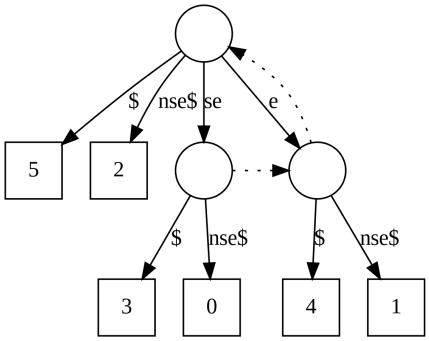

Figure 3.2: Su x tree for the string sense$

3.2 Ukkonen's Algorithm

Ukkonen's algorithm is an online algorithm that builds the su x tree, traversing the string character by character from left to right. Let $! = c_i c_{i+1} :::: c_n$ be a string over some alphabet . The core idea behind Ukkonen's algorithm is building $STree(c_{i+1})$ from $STree(c_i)$ and so forth until the end of the string. The algorithm does this by traversing the *boundary path*.

Definition 2.3.1. A *boundary path* is the path that starts from the deepest state and ends up at the root following the su x links. Boundary path can be seen in gure 3.4.

Definition 2.3.2. An *open transition* is the transition that leads to a leaf state $s \ 2 \ F$. In Ukkonen's algorithm, while building su x tree for string $!$, every leaf state x's reference pair is $(a; (y; 1))$, where the end index is updated to $j!j \quad 1$ when the tree is built.

Definition 2.3.3. An *active point* is the rst state that is not reached by an open transition along the boundary path.

Definition 2.3.4. *End point* is the state that already have x transition when x is being inserted in the su x tree.

To understand how Ukkonen's algorithm works, let's illustrate the working of the algorithm with an example, let $W = c_1 c_2 :::: c_{i \ 1} c_i$ be a word over the alphabet and $STree(c_{i \ 1})$ be the tree we currently have. To extend $STree(c_{i \ 1})$, the algorithm traverses the *boundary path* by adding a c_i transition to each of the states it encounters along the path. However, there are two main groups of states to which c_i transitions must be added as *Lemma 1* [8] states. The one group consists of states reached by open transitions and the other group consists of states starting from the *active point* until

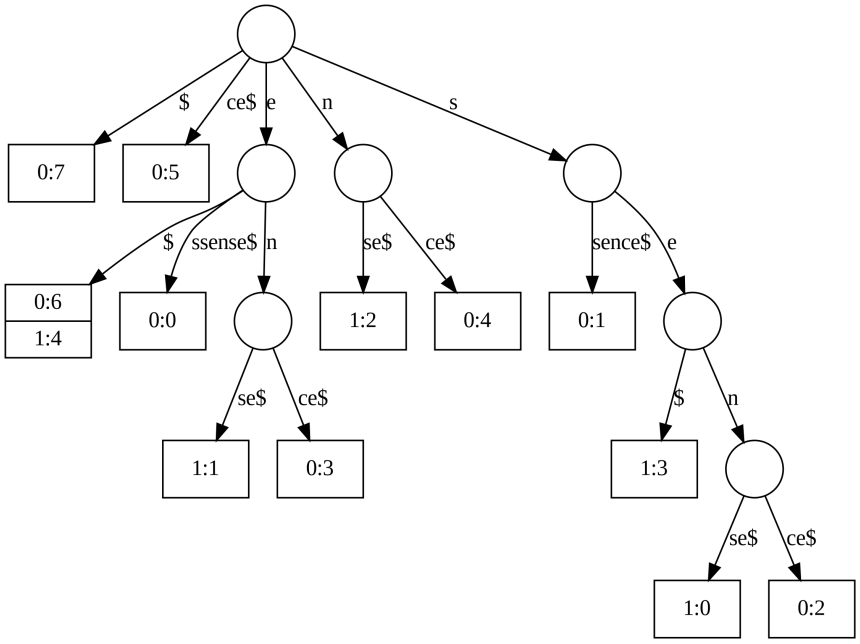

Figure 3.3: Generalized su x tree for the string essence$ and sense$

the *end-point*.

- **Group 1 Open transitions states** : The transition leading to this state is an open transition and we just append the end index of the reference pair for this state.
- **Group 2 States from Active point till end point** :
  - **Case 2a** : If the next character on the edge is c_i then adjust the su x link for the next extension.
  - **Case 2b** : Make the current state explicit if it is not explicit by branching the current transition in two transitions. One transition leading to the old transition and another transition for new open c_i transition. Following that, adjust the su x link for the next extension.

Definition 2.3.5. An *explicit su x* tree for a string *!* over alphabet is a su x tree that contains all the su xes of *!* explicitly. In Laymen's term, there is a path from the root to a leaf for every su x of *!*.

Definition 2.3.6. *Canonical reference pair* : Let $(r;(s;e))$ be the reference pair of the state w then $(r;(s;e))$ is canonical if r is the closest explicit ancestor of state w and s is the updated accordingly but e remains the same.

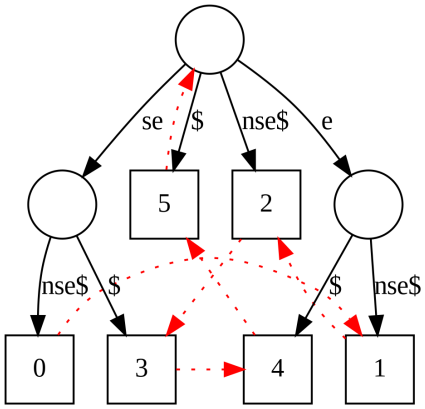

Figure 3.4: Su x tree of string sense with boundary path visible in red dotted arrows

Each string *!* that must be added be terminated with a unique terminal character to ensure the construction of an explicit su x tree. The reason that a unique terminal character say $ makes a tree explicit is because it enforces the addition of $ transition, thereby ensuring the explicit presence of each su x of the string *!*. The above steps are repeated by making sure that the reference pair to the states are canonical before adding the next transitions. The complete Ukkonen's algorithm in pseudocode can be seen down below. In our implementation, *canonize* and *update* are both combined in to a single procedure called *run_phase*. The run time complexity as stated in [8] is $O(n)$, where is the length of the string *!* for which the su x tree is being built.

Algorithm 1 Ukkonen algorithm, where *!* is string over some alphabet and $ is terminal character

```
1: procedure UKKONEN(root, !$)
2:     for i 2 ! do RUN_PHASE(i)
3:     end for
4:     dfs to add leaf state indices
5: end procedure
```

Algorithm 2 The procedure that extends the tree with *i*

```
1: procedure RUN_PHASE(i 2 [ f$g)
2:     while there exists a su x to be added do
3:         if active point is implicit state then
4:             if active point not canonical then
5:                 canonize active point
6:                 continue
7:             end if
8:             if the next character on transition is i then
9:                 update active point
10:                break
11:            end if
12:            create new inner state with transition i.
13:            update active point
14:            update su x link
15:        else
16:            if transition i exists then
17:                update the active point
18:                break
19:            end if
20:            add transition i to the active point
21:            update active point
22:            update su x link
23:        end if
24:    end while
25: end procedure
```

Chapter 4

Generalized suffix tree algorithm

This chapter describes the algorithm that we use to construct a generalized su x tree. Moreover, this chapter also includes the algorithm that is used to extract the most occurring substring of minimum length n from a given generalized su x tree.

4.1 Generalized suffix tree algorithm

This section explains the algorithm used to generate a generalized su x tree. The algorithm used is an extension of Ukkonen's algorithm for on-line construction of su x tree [8], this algorithm was explained in Chapter 3.

Before we jump to our approach, our approach combines the two methods suggested in Gus eld's book [5]. The rst approach concatenates all the strings $!$ with di erent terminal character t and run Ukkonen's algorithm described in algorithm 1 in chapter 3. However, this tree contains su xes that occur in-between two strings. The second approach, that is immune to this issue, is to add each string $!$ terminated with a terminal character t one after the other to the tree by rst running down the tree until a mismatch occurs and then running Ukkonen's algorithm from the mismatched character. The issue with the second approach is that the labels in the transitions may correspond to di erent strings, thus the labels of transitions have to store the string that resulted in the creation of the transition. The approach we assume in this thesis is the combination of the two methods described in Gus eld's book [5].

The basic idea behind the algorithm is to concatenate all $(!_0; !_1; \ldots\ldots; !_n)$ n strings using di erent unique terminal characters $(t_0; t_1; \ldots\ldots t_n)$ for each string $!_i$; $0 \quad i \quad n$ to get $= !_0 t_0 !_1 t_1 \ldots !_n t_n$. Following this, we construct a su x tree for the string . However, this method requires the removal of the su xes that occur between the strings, as a generalized su x tree

should only contain su xes of the strings $!_i; 0 \quad i \quad n$. Therefore, some transitions need to be pruned.

There are two questions that need to be answered in order for us to understand the need for the unique terminal characters and the removal of the in-between strings occurring su x that we do not require our tree to store. These questions are as follows :-

4.1.1 Are N different terminal characters for N strings required ?

The idea behind requiring a unique terminal character is to make the current su x tree explicit. Let $= !_1 t_1 !_2 t_2$ be a concatenation of two strings with unique terminal characters t_1 and t_2 respectively. After string $!_1 t_1$ has been added to the tree we have all su xes of $!_1$ in the tree explicitly and the *root* is the *active point*. We then proceed to add the second string $!_1 t_2$. We can take two things away from the above mini run-through of the algorithm and they are as follows :

- We need at-least one unique terminal character.
- We need to start at root for adding the next string.

These two above points do pose the following question. How does the tree know that a character x is unique to it?

The *root* of a tree has x transitions $x \quad 0$. All these transitions must be unique. If there is a transition $y\ 2$ from the *root* of the tree then the character y is not unique to the tree.

Since Ukkonen's algorithm uses the *boundary path* to add the transitions, this implies that the *root* gets the new transitions at the end. Therefore, if we skip adding the unique terminal character t transition to the root of the tree, t always remains unique to the tree. Consequently, for n di erent strings we do not require n di erent unique terminal characters.

4.1.2 How to avoid adding the in-between strings occurring suffixes into tree ?

In Ukkonen's algorithm, the end index of all leaf states x are updated only after the tree is built. This works perfectly in Ukkonen's algorithm because we are dealing with only one string $!$ which consequently have only a single end index which is $j!j \quad 1$. However, when dealing with multiple strings, we have multiple end indices in the leaf states of a generalized su x tree. Moreover, the *active point* is not always the *root* at the start of adding the su xes of the next string. This happens when we add a su x to the tree that is already present in the tree. To account for these cases, we do the following :

- Since each string is separated by a terminal character *t*, we can update the end index of a leaf state when we reach the end of a string in the run of the algorithm.
- We reset the *active point* back to *root* for adding the next string.

After applying the following changes the pseudocode for the extension of Ukkonen's algorithm for generalized su x tree looks is given below. The RUN_PHASE(*i 2 [f$g*) procedure remains mostly the same with just an addition of the following code after line 15.

```
1: if active point is root and i is $ then
2:     break
3: end if
```

Algorithm 3 Ukkonen, where is a string over some *[f$g* and $ is terminal character

```
1: procedure UKKONEN(root, )
2:     for i = 0;1;:::;j  j  1 do
3:         if i  1 and [i  1] = $ then
4:             update the end index of leaf states
5:             reset active point
6:         end if
7:         RUN_PHASE(i)
8:     end for
9:     dfs to add leaf state indices
10: end procedure
```

4.1.3 Complexity analysis of algorithm 3

We divide the complexity analysis of Algorithm 3 in two following parts.

Lines 1 - 8

The input to the generalized su x tree algorithm and we construct the tree traversing each character from left to right using Ukkonen's algorithm. Therefore, run-time complexity of lines 1-8 of the Algorithm 3 is $O(j\ j)$.

Line 9

We know that there are *n* leaf states for a su x tree for a string *!* of length *n*. Since the number of internal states *i* increases the number of leaf states *n* by at least 1, we can say that $n \quad 1 \quad i$. We also know that there is only 1 incoming transition to every state except the *root*, so this means that the total number of transitions *t* is equal to the sum of *i* and number of leaves

n. Therefore, $t = n + n \quad 1 = 2n \quad 1$. There are $2n \quad 1$ transitions and $2n \quad 1$ states other than the *root*. The total number of transitions and states in a su x tree built for a string $!$ of length n is $2(2n \quad 1) + 1 = 4n \quad 1$.

Similarly, a generalized su x tree built for will have at most $j \ j \quad n$ leaf states. Repeating the above calculation for a generalized su x, we conclude that the total number of states (including *root*) and transitions for a generalized su x tree built for is $4(j \ j \quad n) \quad 1$.

All things considered Algorithm 3 in the worst case have a run time complexity of $O(4(j \ j \quad n) \quad 1)$.

4.2 Most occurring substring of minimum length n algorithm

We are given a generalized su x tree from which we must extract the most occurring substring of minimal length n. Let's break down the problem statement i.e. most occurring substring of minimum length n by rst extracting the most occurring substring regardless of the length and then following that focusing on the length of the substring. As previously done throughout this thesis, we rst analyze the problem statement using a simple construction of a generalized su x tree which is a su x tree.

The most occurring substring on a su x tree is found on a state $x \in root$ that is ancestor to the most leaf states. This is the case because we only create inner states when at-least two sub strings having common pre x. Therefore, the most occurring substring on a su x tree is found on the transition leading to such state x. This can be easily extracted by traversing the tree bottom-up using depth- rst-search and keeping a frequency count on each state.

To extend this approach to a generalized su x tree, we need to take into account that two sub strings can have a common path and hence traversing this path once, as is the case with a simple su x tree, does not consider that the prospect of this path occurring twice in the string . Consequently, we need to traverse the tree multiple times to get the correct frequency count of each state. To get the correct frequency count, we traverse every valid su x present in . In that way, we account for the correct frequency count of each state.

Now that we have the frequency count on each state, we can easily extract the most occurring substring of minimum length n by nding the transitions that lead to the states that occur the most frequent and that are of at least of length n. We can then extract the substring using the start and end index of the transition that leads to these most frequent states. The resulting algorithm is given Algorithm 4. Algorithm 4 can be extended to extract most occurring substrings that occur at least r times the most occurring substring. The value of r ranges from 0:1 to 0:9, therefore $0{:}1 \quad r \quad 0{:}9$.

Algorithm 5 is the algorithm that extracts most occurring substrings.

Algorithm 4 Most occurring substring of minimum length *n*

```
1:  DFS to set occurrences count for every state
2:  node    root
3:  vector    fg        . stores a pair of start index and length of substring
4:  string_dept    0                  . stores the transition depth so far
5:  highest_frequency    0             . keeps track of most frequent state
6:  procedure MOST_FREQUENT_SUBS(node, vector, string_dept, highest_frequency, n)
7:      for i 2 node_transitions do
8:          MOST_FREQUENT_SUBS(i, vector, string_dept + i.transition_depth, highest_frequency, n)
9:      end for
10:     if string_dept < n or (node is leaf state and string_dept == minimum) then
11:         return
12:     end if
13:     length    node:isLeaf ? string_dept    1 : string_dept
14:     if node.occurrence_count == highest_frequency then
15:         vector.push(fnode.end_index - string_dept + 1, lengthg)
16:     end if
17:     if node.occurrence_count > highest_frequency then
18:         vector    fg
19:         highest_frequency    node:occurrence_count
20:         vector.push(fnode.end_index - string_dept + 1, lengthg)
21:     end if
22: end procedure
```

4.2.1 Complexity analysis of Algorithm 4 & Algorithm 5

Since both the algorithms are dominated by the same part of the algorithm in terms of complexity, we just analyze the time complexity of one of the algorithms, namely Algorithm 4. We divide the complexity analysis of Algorithm 4 in two following parts.

Line 1

As mentioned earlier in this document that a single traversal of the generalized su x tree for does not su ce to set the occurrences count of the states accurately. There are at most *j j n* su xes in the tree because there are at most *j j n* leaf states. Traversing a su x of length *m* in a generalized su x tree takes at most *m* steps. Therefore, traversing all *j j n*

Algorithm 5 Most occurring substrings of minimum length *n*

```
1: DFS to set occurrences count for every state
2: node  root
3: vector  fg  . stores a pair of start index and length of substring
4: string_dept  0  . stores the transition depth so far
5: highest_frequency  0  . keeps track of most frequent state
6: procedure MOST_FREQUENT_SUBS(node, vector, string_dept, highest_frequency, n, r)
7:     for i 2 node_transitions do
8:         MOST_FREQUENT_SUBS(i, vector, string_dept + i.transition_depth, highest_frequency, n, r)
9:     end for
10:    if string_dept < n or (node is leaf state and string_dept == minimum) then
11:        return
12:    end if
13:    length  node:isLeaf ? string_dept  1 : string_dept
14:    if node.occurrence_count == highest_frequency then
15:        vector.push(fnode.end_index - string_dept + 1, lengthg)
16:    else if node.occurrence_count > highest_frequency then
17:        remove all substrings whose occurrence count is less than r highest_frequency from vector
18:        highest_frequency  node:occurrence_count
19:        vector.push(fnode.end_index - string_dept + 1, lengthg)
20:    else
21:        if node:occurence_count  r  highest_frequency then
22:            vector.push(fnode.end_index - string_dept + 1, lengthg)
23:        end if
24:    end if
25: end procedure
```

su xes will take m (j j n), where m is the length longest su x present in the tree. All in all, the run time complexity of Line 1 is $O(m$ (j j n)).

Line 2 - 25

The procedure does a simple one pass depth rst traversal of the generalized su x tree of a string of length n. Therefore, the run-time complexity of the procedure is the same as the one discussed in Section 4.1.3 which is $O(4(j$ j $n)$ 1).

All in all, the worst case run-time complexity of the algorithm is $O(m$ j $j)$.

Chapter 5

Codewords case study

Besides the ASML models to test our hypothesis, we would like to test our hypothesis using smaller Mealy machine that also exhibit the far-out distinction problem. This chapter provides in detail the family of Mealy machines we have come up with that exhibit the far output distinction problem.

5.1 Codewords family

This section will detail how the states and the transitions in the codewords family of Mealy machine are formed. Imagine we are at an arcade and we stumble on this machine that has a nite set of prizes P, and in order to win a prize $p \in P$, we must guess the codeword $c \in C$ from the set of codewords C mapped to the prize.

Codeword c for a prize p is a sequence of inputs from I. Let $f : P \to C$ be a bijective function that maps prizes in P to codewords in set C. We also impose a condition on the set of codewords such that a codeword of one prize p is not the pre x of another code-word for another prize p'. Hence, the set C is a so-called pre x code such that for $b; b' \in P$: if $f(b)$ *is a pre x of* $f(b')$ *then* $b = b'$.

There also exists a special input *prize* in I, that tells us the prizes we have won so far. The outputs of this Mealy machine on every input i other than the special input *prize* is -. The output for input *prize* is the set of prizes won so far.

The states of these Mealy machines consists of a tuple $(s; w)$, where $s \subseteq P$ and $w \in I$ such that w is a pre x of a word in C. The number of states in this Mealy machines can be calculated as follows:

- The di erent number of subsets of s is power set of $P = 2^{|P|}$.
- Let the number of pre xes of codewords in C including the empty pre x be $= q$.

- The total number of states is given by $q{:}2^{jPj}$.

Similarly, the number of transitions is calculated as follows:

- The number of states as given above = $q{:}2^{jPj}$.
- The number of inputs is jIj.
- The total number of transition is $q{:}2^{jPj}{:}jIj$

For this class of Mealy machines, we have $I = f0;1;:::;9g [fprizeg$. Therefore, the number of transitions becomes $11q{:}2^{jPj}$.

Let S be the set of states of this Mealy machine. The transitions function $: S \quad I \; ! \; S$ is de ned as follows : for $s \; 2 \; S$, $x \; 2 \; I$, $b \; 2 \; P$, $w \; 2 \; I$ and denotes the empty sequence.

$$((s;w);x) = \begin{cases} (s;wx); & \text{if } wx \text{ is proper pre x of codeword } f(b), \text{ for some } b \; 2 \; P \\ (s \; [\; b; \;); & \text{if } f(b) = wx, \text{ for some } b \; 2 \; P \\ (;; \;); & \text{if } w = \text{ and } x \text{ is not a pre x of any codewords} \\ (s;w); & \text{otherwise} \end{cases}$$

5.2 Motivation

In the gure 5.1, we can see a Mealy machine for two prizes, whose code words are 11 and 22. For the clarity of the gure, p is used instead of the special input *prize* and the prizes won are represented in the form of binary sequence where each index corresponds to each prize, for example 00 represents no prize won and 01 represents second prize won. From the gure itself, we can make up why this is family of Mealy machines exhibit the far-output distinction behavior problem for Mealy machine with large amount of prizes.

The paths of interest in this family of Mealy machines are the paths that lead to all prize winning state which labeled as 9 in gure 5.1. This state can be reached via two paths namely, 0-2-3-4-9 and 0-1-6-8-9. The number of such paths is just the permutation of codewords, where the number of elements and sample size are equal. Hence, for n codewords there are $n!$ such paths that lead to a winning state. In this small example, the lengths of these paths and the number of such paths is small that the learning algorithm can easily learn them. Therefore, since the factorial function grows rapidly, the active automata learning oracle will have di culty learning the Mealy machines of this family with large amount of prizes.

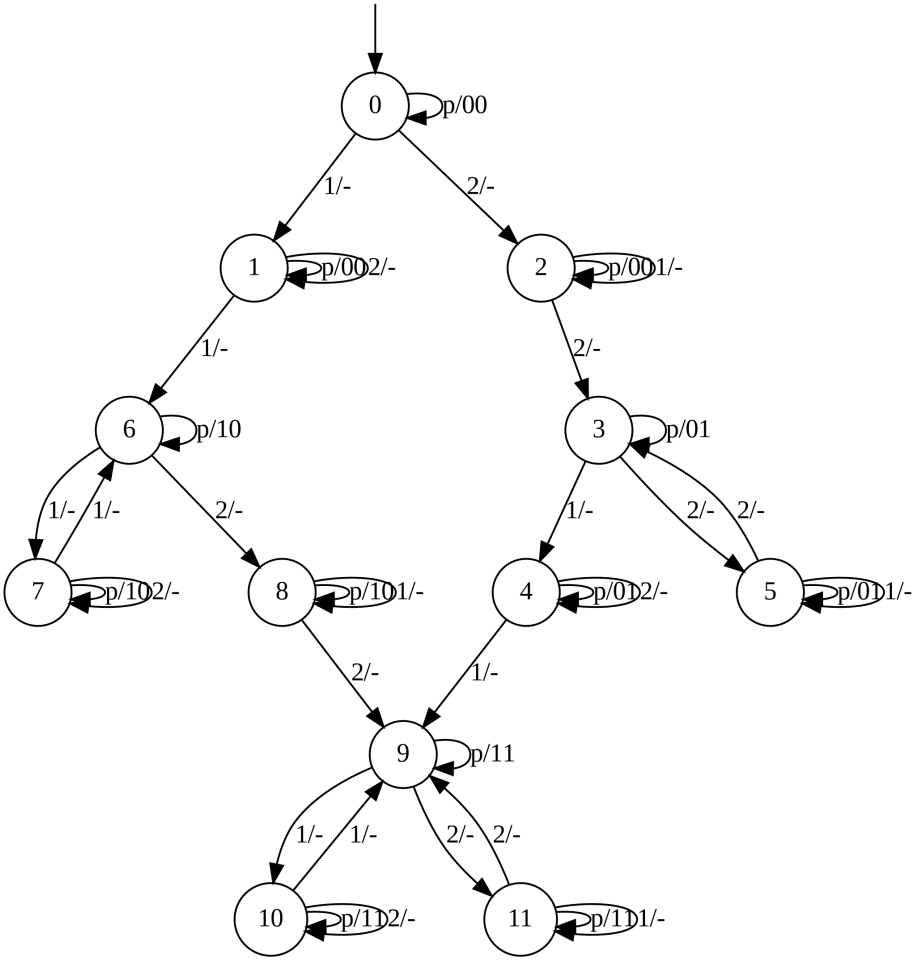

Figure 5.1: A Mealy machine for two prizes with codewords 11 22.

Chapter 6

Experimental Results

This chapter starts with a description of the environment used to carry out the experiments. Furthermore, it presents the results of these experiments.

6.1 Experiment environment

We will first begin by explaining the schematic view of the environment that is used to carry out the experiments. We will test our hypothesis against the codewords case that we introduced in Chapter 5.

Figure 6.1 depicts the environment from top to bottom, starting at the top where codewords for the prizes and a positive integer n are supplied to a script that generates a model (Mealy machine) (labeled as model in Fig: 6.1) for the respective codewords and a traces file that contains n traces that end up in a winning state.

We build a generalized suffix tree (GST) using the generated traces file and a positive integer m which is the minimum length of the most occurring subsequences we want to extract from GST.

We then proceed to feed the most occurring subsequences (SS) and the model to the learning oracle that consists of SUL, learner and the tester. In the experiments conducted, we use the $L^{\#}$ [9] learning algorithm.

6.2 Traces for the codewords models

The script that produces the traces file is fed with an integer n and the codewords. Each trace (TR) is an iteration of a junk (J) trace and a random prize winning trace (RPWT). J traces are valid traces of the model which may or may not win a prize. RPWT are traces for prizes selected at random. The length of a J trace is at most 10 and it is randomly chosen. The total length (TRL) of TR is determined randomly using a random number generator. Let $codeword_i$ be the i^{th} codeword and x be the number of codewords supplied to the script then :

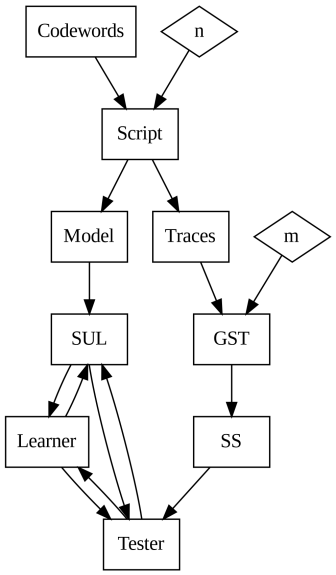

Figure 6.1: Schematic view of experiment process

$$\sum_{i=1} |codeword_i| \quad TRL \quad 10 \sum_{i=1} |codeword_i|$$

TR may or may not win all the prizes, hence at the end of the TR the traces for remaining prizes that are not won are added.

6.3 Experiment setting

We experiment on the models of the codewords in two settings, namely with subsequences (WSS) and without subsequences (WOSS). The di erence in both the settings is in the way test suite T is constructed. The test suite for WOSS is $T = A \; I^k \; C$ and the test suite for WSS is $T = A \; I^k \; S \; C$. Therefore, a test in the WOSS setting has a pre x from A, rst in x from I^k, second in x from S and a su x from C. The sets $A; I^k; S; C$ are set of access sequences, set of permutations of input symbols up to length k, set of SS and set of characterization sequences C respectively. We experimented on both settings with $k = 0$. We also experimented on WOSS setting with $k > 0$ to compare the number of equivalence queries with that of the WSS setting.

6.4 Characteristic of codewords models

We chose the models of codewords to be uniform in terms of the length of each codeword. The choice for this was to reduce the size of the S. We chose the length of codewords l to be at least 4 and the number of codewords c to be at least 2. We then gradually increase l and c to get more complex models. We wanted these models to be in the sweet spot between not too complex and not too trivial. This was mainly to do with the time frame of this project. As such, the models have $l \in \{4;5;6;7;8\}$ and $c \in \{2;3;4\}$.

6.5 Results

In this section, we discuss the results of the experiments we ran. Each experiment consisted of learning the codewords model in WSS and WOSS setting. We set $k = 0$ in the WSS setting. The following two discusses the results when $k = 0$ and $k > 0$ in the WOSS setting.

6.5.1 $k = 0$

For $k = 0$, the WOSS setting failed to learn any of the models given in Table 6.1. The WSS setting learned all the models given in Table 6.1 and as such the length of the SS used is reported under column s of Table 6.1. The SS that are used are presented in Table 6.4. We set s to be the length of the codewords, so that the GST can extract the codewords that were used to construct the model.

6.5.2 $k > 0$

For $k > 0$, we were able to learn some of the smaller models as presented in Table 6.3. Since these models were also learned in the WSS setting, we decided to report the number of membership and equivalence queries that were queried in both the settings as presented in Table 6.3 and Table 6.2. We also reported the number of queries for some of the more complex models in WOSS setting because in comparison with the WSS setting, they were immense. Despite the large number of queries, the WOSS setting still failed to learn the models. Figure 6.2 shows the equivalence queries with respect to the length of codewords in both the settings.

6.6 Conclusion

From the experiments we have conducted, we can say that the knowledge of the most occurring subsequences do de nitely help us in learning models that belong to the codewords family. Lastly, we conclude that the addition of the most occurring input subsequences in to the test sequences de nitely

Model name	*# states*	*# traces*	*l*	*c*	*codewords*	*s*
4_2_20	28	20	4	2	1234 5678	4
4_3_50	80	50	4	3	1234 5678 9438	4
4_4_75	208	75	4	4	1234 5678 9438 2165	4
5_2_20	35	20	5	2	12345 67892	5
5_3_50	103	50	5	3	12345 67892 43716	5
5_4_75	271	75	5	4	12345 67892 43716 51794	5
6_2_20	43	20	6	2	123459 678924	6
6_3_50	127	50	6	3	123459 678924 437168	6
7_2_20	51	20	7	2	1234593 6789244	7
7_3_50	151	50	7	3	1234593 6789244 4371681	7
8_2_20	59	20	8	2	12345936 67892442	8
9_2_20	67	20	9	2	123459361 678924429	9

Table 6.1: Codewords models for experiment

help us reach states that were not reached prior to the inclusion of these sequences. Setting $s = l$ does extract all the codewords of a said model from the GST. Therefore, the findings of the experiment conducted do claim in support of our hypothesis which states that the inclusion of the most occurring subsequences do help with learning models that exhibit the far output distinction behavior.

Model name	*Membership queries*	*Equivalence queries*	*Learned*
4_2_20	980	115	Yes
4_3_50	3921	4963	Yes
5_2_20	1642	41	Yes
5_3_50	6077	4733	Yes
6_2_20	2238	174	Yes
7_2_20	2852	2142	Yes
8_2_20	3826	240	Yes
9_2_20	4458	485	Yes

Table 6.2: Membership & Equivalence queries in WSS setting

Model name	*Membership queries*	*Equivalence queries*	k	*Learned*
4_2_20	920	6859	2	Yes
4_3_50	3748	41715	2	Yes
5_2_20	1541	18876	2	Yes
5_3_50	5192	1193855	3	Yes
6_2_20	1627	493556	3	No
7_2_20	1719	613348	3	No
8_2_20	2371	63608123	5	No
9_2_20	2508	59770207	5	No

Table 6.3: Membership & Equivalence queries in WOSS setting with $k > 0$

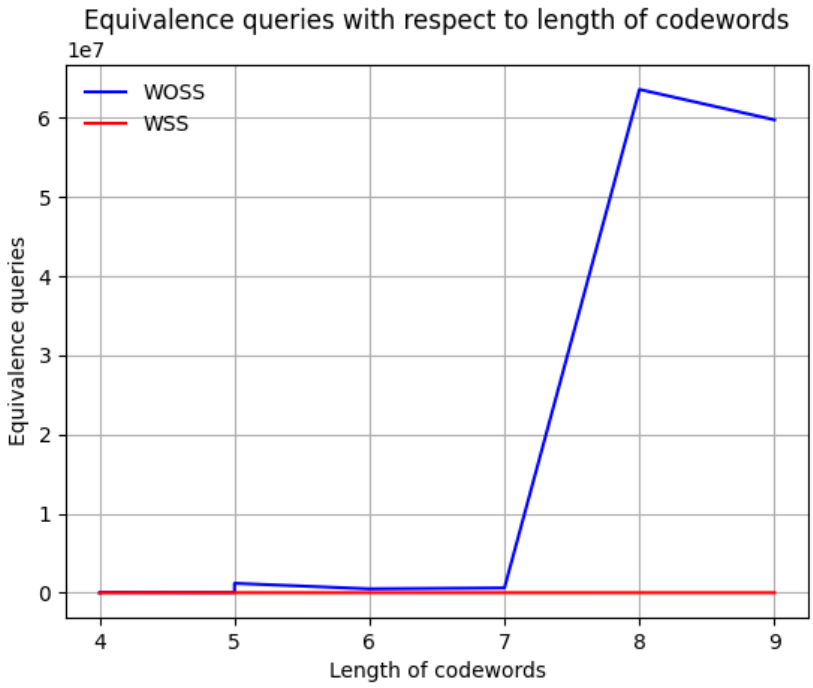

Figure 6.2: Equivalence queries with respect to length of codewords

Model	*Most occurring subsequences*
4_2_20	1234 45678 6781 6781234 2345 781234 5678 81234
4_3_50	1234 12345 81234 2345 234567 2345678 34567 345678 4567 45678 9438 5678
5_2_20	12345 23456 212345 78921 789212345 67892 89212345 9212345
5_3_50	43716 4567892 67892 34567892 12345 243716 23456 234567892 56789 567892
5_4_75	1651794 12345 51794 567892 651794 67892 23456 234567 234567892 243716 34567 34567892 37165 371651 371651794 4567892 43716 437165 71651 71651794 78924
6_2_20	59678924 8924123459 678924 6789241 678924123459 123459 789241 78924123459 9678924 924123459 24123459 4123459
6_3_50	678924 437168 459678924 3459678924 9678924 8123459 234596 23459678924 59678924 123459 1234596
7_2_20	6789244 7892441 7892441234593 892441234593 92441234593 2441234593 441234593 41234593 1234593
7_3_50	5936789244 6789244 936789244 11234593 1234593 12345936 123459367 45936789244 4371681 2345936 23459367 2345936789244 2345934 36789244 3459367 345936789244
8_2_20	12345936 44212345936 4212345936 212345936 244212345936 9244212345936 67892442 89244212345936 789244212345936
8_3_50	23459366 234593667 234593667892442 3667892442 34593667 34593667892442 667892442 67892442 43716819 4593667892442 93667892442 12345936 593667892442
9_2_20	9123459361 123459361 678924429
9_3_50	123459361 1678924429 234593614 234593616 23459361678924429 345936167 3459361678924429 361678924429 437168194 459361678924429 59361678924429 9361678924429 61678924429 678924429

Table 6.4: Most occurring subsequences

Chapter 7

Conclusions & Future work

In conclusion, we explored the idea of extending test sequences with the most occurring subsequences to help us learn models that exhibit the far output distinction behavior. We de ned and speci ed a family of Mealy machines called codewords that exhibit the far output distinction behavior. We obtained the set of most occurring subsequences by building a generalized su x tree of the input sequences that we obtain from the logs of these models. For the codewords family of Mealy machine, we generated valid traces of the models which were used to generate the generalized su x tree. We used an extension of Ukkonen's algorithm, proposed by Gus eld [5], to build a generalized su x tree. We ran experiments on the codewords family of Mealy machine in two settings. In the rst setting, we tried to learn the models without the inclusion of the most occurring subsequences in the test sequences. Conversely, in the second setting we included the most occurring subsequences in test sequences. We found the second setting required much fewer queries to learn the models. Therefore, we conclude that the inclusion of the most occurring subsequences in test sequences help with learning of models that exhibit the far output distinction behavior.

7.1 Future works

In this thesis, we experimented our method with the codewords family of Mealy machines. The number of input symbols in these models is capped at 10. This makes them easier to experiment with it compared to the ASML models. Moreover, the knowledge of long input sequences that cause the far output distinction behavior gives an upper-hand in nding appropriate candidates for the length of subsequences to utilize. However, this is not the case with the ASML models. The application of our method to learn the ASML models can possibly help with learning of the model or help expose new behavior that these models exhibit.

Bibliography

[1] Fides Aarts, Joeri de Ruiter, and Erik Poll. Formal models of bank cards for free. In *Sixth IEEE International Conference on Software Testing, Veri cation and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 461{468. IEEE Computer Society, 2013.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87{106, November 1987.

[3] Kousar Aslam. *Deriving behavioral speci cations of industrial software components*. PhD thesis, Eindhoven University of Technology, June 2021. Proefschrift.

[4] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. Learning fragments of the TCP network protocol. In Frederic Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*, volume 8718 of *Lecture Notes in Computer Science*, pages 78{93. Springer, 2014.

[5] Dan Gus eld. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[6] Mathijs Schuts, Jozef Hooman, and Frits W. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In Erika Abraham and Marieke Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 311{325. Springer, 2016.

[7] Bernhard Ste en, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In Marco Bernardo and Valerie Issarny, editors, *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*, pages 256{296. Springer, 2011.

[8] Esko Ukkonen. On-line construction of su x trees. *Algorithmica*, 14:249{ 260, 1995.

[9] Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wi mann. A new approach for active automata learning based on apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 223{243. Springer, 2022.