Bachelor's Thesis Computing Science

Implementing L# Using Radix Trees

Karolína Trgalová
s1113950

June 30, 2026

First supervisor/assessor:
Dr. Frits W. Vaandrager

Second assessor:
Dr. Jurriaan C. Rot

Radboud University

Abstract

In the active automata learning framework, an algorithm infers the model of a black-box system by interacting with it. This thesis presents a version of $L^{\#}$, a novel active automata learning algorithm, that incorporates the data structure of radix trees. This new algorithm, named $L^{\#}$-radix, integrates with the Python AALpy library in order to maximize accessibility and ease of use. We evaluated the memory usage of our version of $L^{\#}$-radix when compared with the original AALpy $L^{\#}$ implementation by running both versions of the algorithm on a set of small benchmark models. Furthermore, we tested the memory usage of an observation tree consisting of a chain of different numbers of nodes, to see how the algorithm would act under the best possible environment for compression. The results of the benchmarks showed insignificant memory savings, however the memory usage of the best case scenario was two-thirds lower in the $L^{\#}$-radix implementation in comparison with $L^{\#}$ when the number of nodes grew large enough. Overall, this suggests that the $L^{\#}$-radix algorithm is most effective when run on very large models.

Contents

Chapter 1

Introduction

Automata learning is the process of understanding the behavior of a system and creating a model that corresponds to it. These models are often represented with finite-state machines. Automata learning can be divided into passive and active learning, where passive learning consists of analyzing runs of a system and active learning, where responses to input queries are evaluated. We will focus on active learning, in which a learner program poses either membership queries (input-output pairs) or equivalence queries (testing the correctness of a current hypothesis) to an oracle, which knows the system. In applications, equivalence queries are often approximated by posing a large number of membership queries. Due to this, equivalence queries in automata learning algorithms often have a high time complexity, and are the largest obstacle in increasing the efficiency of such algorithms [1].

There are a variety of active automata learning algorithms, all using different approaches and offering various benefits. L^* is a well-known example, first developed by Angluin in 1987 [2]. This algorithm organizes the results from membership queries into an observation table, and uses this information to construct equivalence queries. The memory consumption of this algorithm is quite high, and can become an obstacle to learning large models [3]. It also suffers from the common problem of ineffective equivalence queries [1].

The $L^\#$ algorithm, first introduced by Vaandrager et. al. is an active automata learning algorithm which attempts to solve the issue of the high complexity of equivalence queries. It uses many of the same concepts as L^*, but instead of working with the equivalence of observations, it focuses on the idea of apartness, or proving that two states are not equivalent, for handling its tree-based equivalence queries. It further provides innovations in reducing the data structures used, simplifying the algorithm [4]. AALpy, a Python library dedicated to automata learning, is one of the most accessible implementations of this algorithm [5].

$L^\#$ works directly on the observation tree, a Mealy machine that represents all of the observations of inputs and outputs on the system [4]. This is the primary data structure that the algorithm uses, and as such is responsible for the majority of the memory consumption. As the observation tree stores the result of every single query [4], the memory consumption can become undesirably high when working with large models [6]. There are a variety of possible solutions for this issue; for instance the TTT algorithm has a redundancy-free organization of observations which enables TTT to have a linear space complexity [7]. We will focus on the implementation of memory-saving data structures [3], such as adaptive radix trees [8].

The adaptive radix tree [8] is a space-efficient memory structure that can replace the traditional observation tree. This memory structure uses the direct representation of keys instead of hashing or comparing them, which allows them to be ordered, a key aspect of a data structure

for our purposes. Furthermore, it chooses how nodes are represented dynamically, depending on the number of child nodes. This allows the adaptive radix tree to collapse nodes and decrease tree height, which significantly improves memory efficiency [8].

This thesis builds on the work of Garhewal, who created a new implementation of $L^{\#}$ in Rust with adaptive radix trees [6]. Although foundational for our research, the choice of programming language and the creation of a wholly new implementation limited the accessibility of this implementation. This motivates the research questions that we will explore in this thesis:

RQ1: Can we rewrite the implementation of the active automata learning algorithm $L^{\#}$ that is compatible with the AALpy Python library while using radix trees to represent observation trees?

RQ2: How does the memory usage of the radix tree version of $L^{\#}$ compare to the existing AALpy $L^{\#}$ implementation when evaluated on a collection of benchmarks?

RQ3: Can we give an estimate of the maximal memory savings that can be achieved by the radix tree version of $L^{\#}$?

The following section explains the $L^{\#}$ algorithm and provides further detail on adaptive radix trees. Next, we analyze how the $L^{\#}$ algorithm in the AALpy library was altered and provide justification for our design choices. The specifics of how the memory usage was tested are described in detail, and the results are charted. Finally, this thesis concludes with a discussion and conclusion about whether adaptive radix trees are applicable to the $L^{\#}$ algorithm.

Chapter 2

Preliminaries

2.1 The $L^{\#}$ Algorithm

The $L^{\#}$ algorithm, first described in Vaandrager et al.'s work [4], is an algorithm for learning deterministic finite state machines (FSM), such as the one in Figure 2.1. This algorithm is based on the idea of query learning, in which a learner obtains information about an automaton by asking questions to a teacher [2]. At first, the learner is only aware of the possible inputs.

The core idea behind this algorithm is *apartness*. Two states q and r are apart ($q\#r$) if there exists an input sequence that results in a different output.

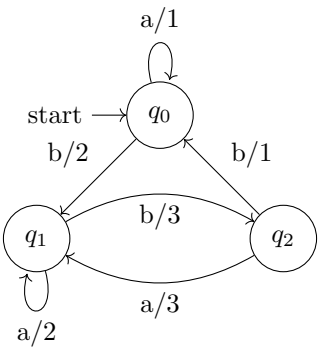

Figure 2.1: The example FSM.

The $L^{\#}$ algorithm stores the information about the FSM it is learning in an observation tree. An observation tree is a partial FSM where each output sequence results in a new state. Each state of the observation tree is equal to one of the states in the target FSM, and the goal of this algorithm is to merge the states of the observation tree in such a way that the resulting FSM will be equal to the one the algorithm is trying to learn. An observation tree corresponding to the FSM in Figure 2.1 is in Figure 2.2.

For the $L^{\#}$ algorithm to work, the states in the observation tree must be divided into three different sets: the basis states S, the frontier states F, and the remaining states, which are not in either set. The basis states consist of the initial state t_0 of the automaton, as well as all other states that that are apart from t_0 and each other. The frontier states are all states that can be reached from S in only one transition.

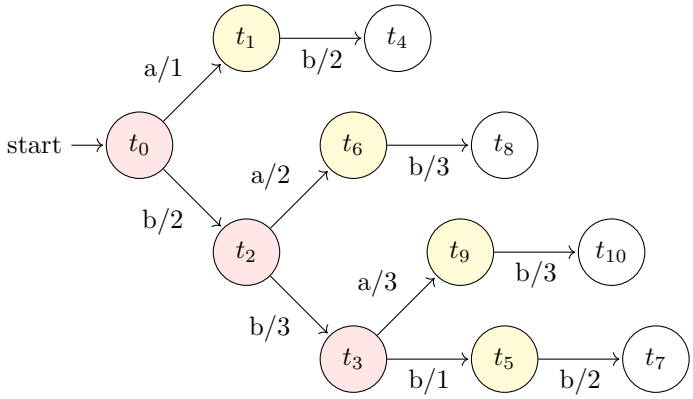

Figure 2.2: The final observation tree of the example FSM. S is in red, F is in yellow.

Each state $q \in F$ has a candidate set $C(q)$. This candidate set is comprised of all possible basis states that q could correspond to, that is, basis states from which it is not apart.

The $L^{\#}$ algorithm functions by employing four rules. These are applied to the observation tree in an arbitrary order as long as their conditions are met.

1. **Promotion** is applied if a state $q \in F$ is apart from all basis states, that is $C(q) = \emptyset$. q is removed from F and added to S.

2. **Extension** is applied if a state $q \in S$ does not have outgoing transitions for all possible inputs. This rule extends q with the missing transitions to the new states.

3. **Identification** is applied if a state $q \in F$ has a candidate set $C(q)$ that contains two or more elements. An input sequence that distinguishes between at least two of them is chosen, and q is extended with this input sequence. In the resulting observation tree, $C(q)$ is reduced by the basis states that have been ruled out.

4. **Equivalence** is applied if none of the other rules can be applied. The algorithm creates a hypothesis of how the FSM could look like based on the observation tree, and asks the teacher whether it is correct. If the answer is yes, the algorithm is complete. If the answer is no, the teacher returns a counterexample that is then further processed.

In order to show how these rules work, the algorithm will be applied to a test case, the FSM in Figure 2.1. This FSM has two possible inputs, a and b, and three possible outputs, 1, 2 and 3.

Initially, the observation tree consists of the initial state t_0, which belongs in S. The rule of **extension** is the only one that can be applied, so t_0 is extended with transitions to t_1 and t_2, as can be seen in Figure 2.3. At this point, no rule other than **equivalence** can be applied. The algorithm creates a hypothesis based on the observation tree, as can be seen in Figure 2.4. Since there is only one basis state, t_0, the candidate sets for both of the other states contain only t_0. The hypothesis is therefore of a single state, with transitions that go to itself.

This example is then posed to the teacher, which returns a counterexample, say b b. The counterexample is processed and the observation tree is extended in Figure 2.5. The processing of the counterexample is quite difficult, however it is beyond the scope of this paper. Further details can be found in Vaandrager et. al.'s paper [4]. As the state t_2 produces a different output for the input b and is therefore clearly apart from t_0, the rule of **promotion** is applied and the

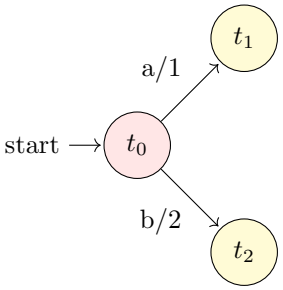

Figure 2.3: The initial application of the extension rule.

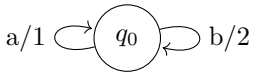

Figure 2.4: The initial hypothesis.

basis becomes $S = \{t_0, t_2\}$. This means that the frontier states of t_1, t_3 have a candidate set of t_0, t_2 and so the **identification** rule can be applied to both of them. The observation tree is extended with the use of a b and b b b as the input sequences respectively, which creates the states t_4 and t_5, which can be seen in Figure 2.6.

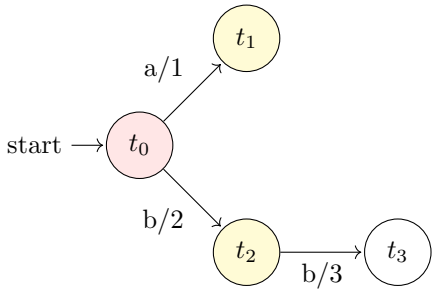

Figure 2.5: The extension of the FSM after the counterexample.

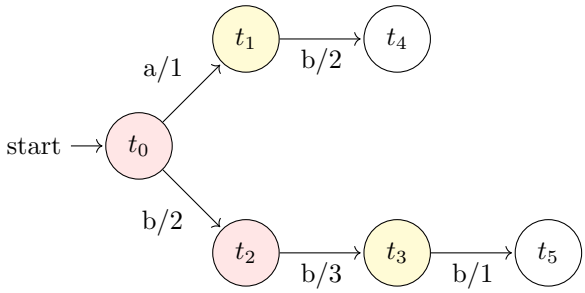

Figure 2.6: The application of the promotion and identification rules.

The process continues with the **extension** rule being applied to t_2, creating t_6. Next, the **promotion** rule is applied to t_3. The **identification** rule is applied to t_5 and t_6, creating t_7

and t_8. t_3 is **extended** with a transition to t_9, to which the **identification** rule is immediately applied to create t_{10}. At this point, the observation tree is equivalent to the one seen in Figure 2.2. The only rule that can be applied is **equivalence**, and so another hypothesis is created. This time, the hypothesis is equivalent to the target FSM, and so the algorithm terminates.

2.2 Adaptive Radix Trees

Large observation trees frequently contain long sequences of nodes with only one child node each, such as the sequence in Figure 2.7. Each node stores a successor dictionary that contains a reference to only one node, which is clearly inefficient.

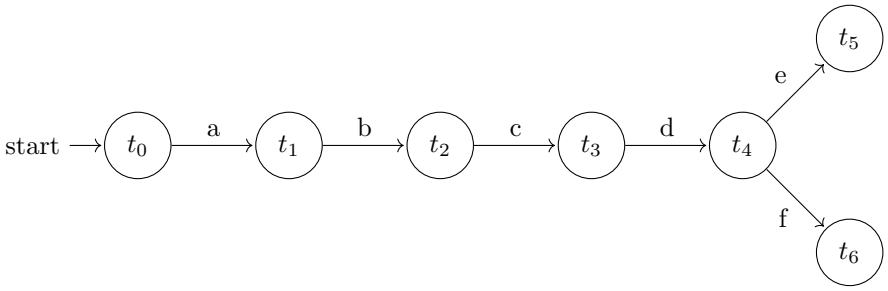

Figure 2.7: An example of a sequence of nodes with only one child each.

We can improve the memory usage of observation trees by borrowing a few techniques from Leis et. al's work on adaptive radix trees [8]. Adaptive radix trees compress nodes to reduce the height of the tree. Additionally, they dynamically choose between various data types for the nodes depending on the number of children.

There are three techniques adaptive radix trees use:

1. **Lazy expansion**, where nodes are only created if they are required to distinguish two leaf nodes. If it is a node that is the only child of another node, it is immediately compressed.

2. **Path compression**, where existing nodes that only have a single child are removed and merged with another compressed node, or changed into a compressed node along with its child.

3. **Adaptive nodes**, where a node's data structure is dynamically chosen based on the number of children it has.

Path compression and lazy expansion are achieved by turning a sequence of nodes into a singular list which contains the transitions between the nodes and a child node, as can be seen in Figure 2.8. This takes up significantly less memory than the original observation tree.

There are four node types in the work of Leis et. al. [8]. Node4, Node16, Node48, and Node256. Each of these can store the number of child pointers and key/value pairs that are in its name. Node256 uses an array of key pointers, where the byte value itself is the array index.

The paper by Leis et. al. elaborates on two alternative ways to implement path compression [8]. There is pessimistic allocation, where a compressed node stores the data it contains, and optimistic allocation, where a compressed node only stores the length of the compressed sequence, and the algorithm skips this length during traversal. Only then is the sequence validated to ensure that it matches. Leis et. al. combine these two approaches into one by using a pessimistic

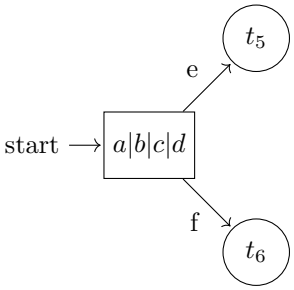

Figure 2.8: A compressed version of Figure 2.7.

approach if a sequence is equal to or shorter than a specified length (8 bytes) and an optimistic approach if a sequence is longer than that.

2.2.1 Adaptive Radix Trees in $L^{\#}$

To implement radix trees into $L^{\#}$, we had to make a few changes to the concept. It was not possible to use the exact method of choosing a node's data structure that Leis et. al. use. Garhewal's Rust implementation of $L^{\#}$ with radix trees shows that this is not ideal, since the input alphabets for the automata that the algorithm learns might be larger than 256 symbols [6]. Instead, inspired by Garhewal's innovative work, our approach to adaptive nodes was to have two node types: an uncompressed node class, nearly identical to the one used in the original $L^{\#}$, and a compressed node class, which stored the critical data of the nodes it contained in a list.

We chose a pessimistic approach to compression, as this fits better with how the original data structure, observation trees, works. In comparison, using an optimistic or mixed approach and only storing the compressed sequences of nodes in a leaf node would require large parts of the program to be rewritten, as there are many operations in $L^{\#}$ that do not traverse the length of a branch until the leaf node.

Furthermore, we only used the lazy expansion concept in compressing our observation tree. As there are no node deletions in an observation tree, which is not guaranteed in a generic radix tree, and each node is inserted one by one, a single node with one child can only be created as a tree grows (which falls under lazy expansion). This allowed us to make the algorithm more time-efficient, since we did not have to check for path compression whenever we updated the observation tree.

We believe that this will lead to memory savings as the memory cost of adding an entire object to an observation tree is much larger than the memory cost of appending an element to a list within an already existing object. The cost of adding an element to a list is approximately eight bytes, while an object of a class with four attributes takes up 64 bytes. This is due to the fact that a new object contains an object header and the internal fields for the attributes, while appending an element to a list does not add that to the memory. This means that the longer the compressed chain of nodes is, the higher the memory savings compared to $L^{\#}$ without compression will be.

Chapter 3

Implementation

In this chapter, we discuss the design choices and explain in detail how the $L^{\#}$ algorithm was adapted to be compatible with the radix tree data structure. It covers the key design choices, the data structure of compressed nodes, and how the compression itself functions.

This thesis is based on the Python implementation of the $L^{\#}$ algorithm in the AALpy library [5]. This version of the $L^{\#}$ library was chosen as it is the highly accessible and often used in other research regarding this algorithm.

3.1 Design Choices

A key design choice was the structure of the compressed node. The Rust version by Garhewal used a *struct* to implement both compressed and uncompressed nodes [6]. The usage of Python necessitated a different choice for our implementation, so we chose a structure that paralleled the already existing nodes. This became the new *CompressedMealyNode* class with multiple attributes and methods, in addition to the original uncompressed *MealyNode* class.

Equally important was the decision of how to handle basis and frontier nodes. Since compressed nodes are not static like uncompressed Mealy nodes, and are prone to being split into multiple nodes, we decided to proactively uncompress frontier and basis nodes to prevent errors. The splitting of a CompressedNode could be problematic if it contained a frontier or basis node, and the data structures that hold these are not updated properly. Although this will increase the memory consumption, the number of basis and frontier nodes is likely fairly small compared to the number of remaining nodes.

3.2 Details of the Implementation

3.2.1 The Data Structure of Uncompressed Nodes

The *MealyNode* class contains attributes for the *id, parent, input_to_parent*, and a dictionary of *successors*, where the key is the input to the next node and the value is a tuple containing the output of the next node and the next node itself. It has methods for adding a successor, returning a successor node for a given input, and returning the output for a given input.

```
class MealyNode:
    _id_counter = 0
    __slots__ = ['id', 'successors', 'parent', 'input_to_parent']
```

```
    def __init__(self, parent=None, id=None):
        if id is None:
            MealyNode._id_counter += 1
            self.id = MealyNode._id_counter
        else:
            self.id = id
        self.successors = {}
        self.parent = parent
        self.input_to_parent = None

    def __hash__(self)
    def add_successor(self, input_val, output_val, successor_node)
    def get_successor(self, input_val)
    def get_output(self, input_val)
    def extend_and_get(self, inp, output, basis, frontier)

    @property
    def id_counter(self):
        return self._id_counter
```

3.2.2 The Data Structure of Compressed Nodes

We created a new class to hold the compressed nodes in. This class has a similar structure to the class for uncompressed Mealy nodes, keeping most of the attributes. The *successors* dictionary was kept to keep track of which nodes the final node in the compressed nodes led to. The only attribute unique to the class is the *nodes* list which contains the critical information of the nodes the compressed node represents. Within the list, each node is represented as a tuple with its ID, output symbol, and the input symbol to the next node. The methods that *MealyNode* contains are preserved, with the addition of functions that get the parent or input to parent at a given index in the *nodes* list. A function that uncompresses one of the nodes is also included.

```
class CompressedMealyNode:
    __slots__ = ['nodes', 'successors', 'parent', 'input_to_parent']

    def __init__(self, nodes, parent=None):
        self.nodes = nodes
        self.successors = {}
        self.parent = parent
        self.input_to_parent = None

    def add_successor_middle(self, input_val, output_val, successor_node, index)
    def add_sucessor_end(self, input_val, output_val, successor_node)
    def get_successor(self, input_val, index)
    def get_output(self, input_val, index)
    def extend_and_get(self, inp, output, index)
    def get_input_to_parent(self, index)
    def get_parent(self, index)
    def uncompress_node_at_index(self,index)
```

3.2.3 Traversing Compressed Nodes

Many functions call for traversing the observation tree node by node by repeatedly calling a function that gets the parent of a node, and then calling the same function on that parent, and so on. This would not work with our compressed node structure, as this sort of function would have to skip past all of the nodes in the compressed *nodes* list.

To solve this issue, many *CompressedMealyNode* methods were augmented with an *index* argument that would determine on which node in the *nodes* list the function should be performed. Additionally, these methods return a tuple with the original output and the *counter*, which increases/decreases by one depending on the exact function, or if it reaches the end/start of the node it is set to zero so the counter variable is correct when a function is performed on another compressed node with it as the value for the index argument. All functions that used these methods had to be changed to also keep track of the counter and to account for the varying inputs between *MealyNodes* and *CompressedMealyNodes*.

A good example of how the implementation of the counter is the *insert_observation* function, which inserts an observation into the tree using sequences of inputs and outputs:

```
def insert_observation(self, inputs, outputs):
    current_node = self.root
    counter = 0
    for input_val, output_val in zip(inputs, outputs):
        if type(current_node) == CompressedMealyNode:
            current_node, counter =
                current_node.extend_and_get(input_val, output_val, counter)
        elif type(current_node) == MealyNode: current_node, counter =
            current_node.extend_and_get(input_val, output_val, self.basis,
            self.frontier_to_basis_dict.keys())
        else: current_node = current_node.extend_and_get(input_val, output_val)
```

3.2.4 Creating Compressed Nodes

If an uncompressed node is extended with a successor node and it has no other successors, and doesn't belong in the frontier or basis, the node immediately changes itself and the successor into a compressed node. The node's parent also cannot be in the frontier or basis, as this would cause issues with updating the *successors* dictionary if the compressed node was somehow changed. As nodes are never removed from the observation tree and they're always added one by one, this method covers all of the possibilities of where a compressed node can be created.

3.2.5 Extending Compressed Nodes

Extending a compressed node results in two cases:

1. A node added to the end of the compressed node.
2. A node added to the middle of the compressed node.

A new node can be appended to the end of the compressed node with minimal changes, as it simply necessitates adding a new element to the end of the *nodes* list. Inserting a new node into the middle of a compressed node is more challenging. The compressed node must be split into two compressed nodes. The portion up to the split is preserved, and the new node is added to

the *successors* dictionary along with a new compressed node which contains the second part of the original compressed node.

Chapter 4

Results

4.1 Method

The goal of this thesis is to compare the memory consumption of $L^{\#}$-radix and the unaltered version of $L^{\#}$ without compression. To achieve this, we decided to test the algorithms on the same benchmarks and measure memory usage. Each experiment was run five times on both $L^{\#}$-radix and the original version of $L^{\#}$. The memory usage was measured using the Python *tracemalloc* library and recording the peak memory usage during the runtime of the model learning.

Eight Mealy machine models were chosen for testing: Gnu-TLS [9], MQTT [9], Passport [9], TCP-Ubuntu [9], Bankcard [9], m41 [10], m54 [10], and m183 [10].

Model	**Number of states**	**Alphabet size**
Gnu-TLS	15	12
MQTT	18	9
Passport	4	19
TCP-Ubuntu	57	12
Bankcard	7	14
m41	25	47
m54	26	10
m183	9	10

Table 4.1: The number of states and alphabet size of each model.

Gnu-TLS is a Mealy machine that models Gnu-TLS in an effort to find security flaws. It is a fifteen-state model with a highly connected sink state that has multiple transitions from every other state. Each state has self-loops, some have multiple. This Mealy machine has a twelve-element alphabet.

MQTT is an eighteen-state model that represents the MQTT protocol using an alphabet size of nine. This heavily cyclical graph has nine transitions per state corresponding to specific client inputs.

Multiple biometric passport protocols are represented by the Passport model. It has four states and an alphabet size of nineteen.

TCP-Ubuntu is a model that represents the software components of the Transmission Control Protocol implementation on Ubuntu. It consists of fifty-seven states and an alphabet with twelve elements, making it the largest benchmark.

The Bankcard Mealy machine was learned from the bankcard EMV protocol as applied at Volksbank. It has seven states, each of which have fourteen uniform outgoing transitions. Two of these are "error states", receiving the majority of incoming transitions.

M41 is a highly branched Mealy machine with twenty-five states and forty-seven elements in its alphabet. It has one "error state" which the majority of the transitions lead to. The remaining network always ends up looping into the initial state.

M54 is a twenty-six state Mealy machine with only ten transitions per state. It also has one error state that most transitions end up in.

M183 is a relatively small nine-state benchmark with an alphabet size of ten. Most of the transitions lead to state two, the designated "error state".

We conducted additional experiments to evaluate memory usage under maximum compression by manually inserting a single chain of nodes into an observation tree containing solely a root node. We then compared the peak memory usage of the original $L^{\#}$ algorithm against $L^{\#}$-radix. To effectively assess the compression's scalability, we varied the chain lengths on a logarithmic scale: 100, 1 000, 10 000, and 100 000 nodes. Each configuration was repeated five times to ensure reliability.

4.2 Results

Our research question was whether we can improve the memory consumption of $L^{\#}$ using radix trees. Furthermore, we were interested in whether $L^{\#}$-radix affects runtime negatively.

The table 4.2 shows the average peak memory usage of $L^{\#}$ and $L^{\#}$-radix when run on the benchmarks. For most of the benchmarks, the memory usage is nearly identical. The differences between $L^{\#}$ and $L^{\#}$-radix are likely simply due to statistical noise, as they are within the standard deviations. During the TCP-Ubuntu experiment, $L^{\#}$-radix used on average 2.56 MB less memory. However, the standard deviation for this benchmark was 4.361, which means that this result is within the error bars. The data for this experiment shows that four out of five of the iterations used around 46 MB, but one of the runs only used 35.3 MB. The next largest standard deviations were 0.621 for the $L^{\#}$-radix version of m41 and 0.185 for the $L^{\#}$ version of TCP-Ubuntu; the rest were smaller than 0.1. The standard deviations of each of the experiments can be found in Appendix B.

This experiment showed that our research question RQ1, whether we can create an AALpy version of $L^{\#}$ that uses radix trees, can be answered positively. All of the benchmarks were able to run successfully on $L^{\#}$-radix.

Furthermore, we were able to answer our research question RQ2, concerning the memory consumption of running benchmarks on $L^{\#}$-radix compared to $L^{\#}$. For this collection of small benchmarks, it appears that the memory usage is nearly identical between the two algorithms.

In order to see how memory usage was affected in the best possible scenario, we measured the peak memory usage of an observation tree that consisted of a root node and an n-long chain of nodes. Each number of nodes was run five times and the results were averaged, which can be found in Table 4.3. The memory usage in this table only represented the observation tree itself, the rest of the program peaked at an average memory consumption of 7.085 MB for $L^{\#}$ and 6.5827 MB for $L^{\#}$-radix. As these measurements were taken before the string of nodes was inserted, they remained consistent across all iterations of the experiment. A chain of 100 nodes was too small to be able to be measured in both algorithms. A string of 1 000 nodes resulted in $L^{\#}$ using 0.2227 MB and $L^{\#}$-radix using 0.0022 MB. This means that $L^{\#}$-radix only consumed 0.98% of the memory that $L^{\#}$ did. When the number of nodes in the observation tree reached 10 000 and 100 000, $L^{\#}$-radix became more memory-efficient than $L^{\#}$ by about a third, utilizing

Model	$L^\#$ **memory usage**	$L^\#$**-radix memory usage**
Gnu-TLS	10.32	10.34
MQTT	1.48	1.50
Passport	0.60	0.60
TCP-Ubuntu	46.56	44.00
Bankcard	0.30	0.30
m41	167.02	167.36
m54	2.68	2.70
m183	1.00	1.00

Table 4.2: Peak memory usage (in MB) of $L^\#$ and $L^\#$-radix while running the benchmarks.

33.527% and 35.331% of $L^\#$'s memory usage respectively.

Our final research question, RQ3, can be answered by this experiment. Under ideal conditions, we estimate that $L^\#$-radix saves around 99% of $L^\#$'s memory.

Number of Nodes	$L^\#$ **memory usage**	$L^\#$**-radix memory usage**
100	0	0
1 000	0.2227	0.0022
10 000	3.3206	1.1133
100 000	34.2123	12.0876

Table 4.3: Memory usage (in MB) of the string of nodes in $L^\#$ and $L^\#$-radix.

We are also interested in how the $L^\#$-radix algorithm performs time-wise when compared to the original $L^\#$, which can be seen in Table 4.4. Most benchmarks perform slightly worse on $L^\#$-radix, with the largest difference being 1.38 seconds on the m41 benchmark. Nonetheless, this only means that the runtime of $L^\#$-radix is 107.93% of that of $L^\#$. On the other hand, the models m54 and m183 show a better performance from $L^\#$-radix. $L^\#$-radix has a runtime of merely 57.524% of $L^\#$ when learning the model m54. The benchmark m183 shows a slightly more modest improvement of 63.38% over $L^\#$.

This experiment showed that on small models, $L^\#$-radix generally has a small negative effect on the runtime. It seems that for some models, $L^\#$-radix can improve the runtime by nearly half, although the reason behind this is uncertain.

Model	$L^\#$ **runtime**	$L^\#$**-radix runtime**
Gnu-TLS	0.48	0.48
MQTT	0.15	0.18
Passport	0.03	0.04
TCP-Ubuntu	5.09	5.53
Bankcard	0.03	0.04
m41	17.71	19.09
m54	0.82	0.47
m183	0.14	0.09

Table 4.4: Runtime (in seconds) of $L^\#$ and $L^\#$-radix while running the benchmarks.

Chapter 5

Discussion

$L^{\#}$-radix does not appear to save memory when run on small benchmarks. Most benchmarks showed no improvement in memory usage for $L^{\#}$-radix. In fact, models that took up more than one megabyte resulted in a slightly increased memory usage for $L^{\#}$-radix. As these differences are equal to or lower than 1% of the overall memory consumption, and are within the standard deviation, it is likely that they result from statistical noise. Even if they were caused by the radix tree data structure, they are small enough to be negligible.

The only outlier was TCP-Ubuntu, where $L^{\#}$-radix observed savings of over two megabytes compared to $L^{\#}$. However, this is not a significant finding, as most of the runs of this experiment did not show a difference between $L^{\#}$-radix and $L^{\#}$. There was only one iteration of this experiment where the difference between the memory consumption of the two algorithms was more than ten megabytes, but this could have been due to chance rather than $L^{\#}$-radix's compression.

When analyzing the broader dataset, there is no evidence to suggest that the specific characteristics of the benchmarks influenced the performance of $L^{\#}$-radix. Other than the single anomalous run of the experiment run on TCP-Ubuntu, the small memory consumption differences seem to result from statistical noise. This lack of variance indicates that for the size of benchmarks that we used, $L^{\#}$-radix performs identically to standard $L^{\#}$.

Consequently, these results suggest that $L^{\#}$-radix provides no memory benefits for small benchmarks. The method of compression that radix trees use requires long strings of nodes with only one child each, which are the most likely to result from long counterexamples to rejected equivalence queries. As the number of states a model has rises, so does the length of the counterexamples. This hypothesis is further confirmed by Garhewal's work on a Rust version of $L^{\#}$-radix, which showed that this algorithm only tends to be effective when dealing with large models [6]. His implementation only had significant memory savings on a benchmark with 546 states, which is far more than the benchmarks we experimented with. As Rust is usually more memory-efficient than Python, it is possible that our implementation might require even larger models to be effective. At the same time, Garhewal's experiments differed in that he also stored the results of test queries in the observation tree. This could mean that our work is less comparable than it might seem at first glance.

In order to see how $L^{\#}$-radix and $L^{\#}$ would behave under a best case scenario for compression, we tested how much memory it would consume when a string of nodes was manually inserted. This allowed us to have an approximation of how $L^{\#}$-radix could behave when confronted with large models. The memory savings for paths of nodes 1 000 long were very significant. $L^{\#}$-radix used only 1% of $L^{\#}$'s memory usage in this case, which amounted to savings of 0.2205 MB on average. This is a significantly higher percentage than the next two node chain sizes of 10 000

and 100 000, where we only observe $L^{\#}$-radix consuming about 34% of $L^{\#}$'s memory usage. We were unable to definitively find out why the 1 000 node long string was significantly more effective than the longer strings, but it is possible that this has to do with Python memory allocation. If Python allocates far more memory than necessary to lists with more than 1000 elements, the longer lists could be less efficient. Further research into this phenomenon is needed. Although savings of 0.2205 MB are not very large by themselves, especially when compared to the nearly seven megabyte overhead, a large observation tree that had many counterexamples of this size inserted would consume significantly less memory if using $L^{\#}$-radix.

The results of this experiments allows us to approximate the memory usage of $L^{\#}$-radix for very large models. If B stands for the baseline memory usage of the algorithm, O stands for the size of the observation tree in $L^{\#}$, and M stands for the memory usage of $L^{\#}$-radix, we get the approximate equation of $M = B + \frac{O}{3}$. Further experiments on large models are required to confirm whether this prediction holds.

Lastly, it is important to discuss the runtime. While an algorithm being more memory efficient is positive, if it takes too long to run it can become very difficult to use. The time efficiency of $L^{\#}$-radix was affected negatively in most cases, however the difference between $L^{\#}$ and $L^{\#}$-radix only goes above one second in the case of m41, which has a notably large alphabet. This suggests that our implementation of $L^{\#}$-radix does not handle large alphabet sizes well runtime-wise. Although these differences are small, if a benchmark was run hundreds of times, it would add up to a significant amount. Due to this, we conclude that our implementation of $L^{\#}$-radix is unsuitable for small models that are unlikely to benefit from the compression, as it would unnecessarily slow them down. As stated before, we were unfortunately unable to run $L^{\#}$-radix on larger models where the differences in runtime could be larger.

$L^{\#}$-radix was nearly twice as fast as $L^{\#}$ while running two models: m54 and m183. As each experiment was only run five times, it is possible that this is due to a statistical anomaly. These two models also share a low alphabet size and a small observation tree, although this decrease in runtime was not observed in other models with low alphabet sizes and small observation trees. Further research is needed to confirm the reason behind this.

5.1 Possible Improvements

The current implementation of $L^{\#}$-radix in AALpy suffers from multiple issues. Nodes that are in the basis or frontier have to be uncompressed, as compressed nodes are inherently less stable and sometimes split apart. There are also several other points at which nodes are unnecessarily uncompressed, as we were unable to get certain functions to run if they were given a compressed node as an argument. Changing the code base so every function and the basis and frontier system are compatible with compressed nodes would very likely improve on the memory savings.

The CompressedMealyNode and MealyNode classes could be merged into one class that natively allows compression. This would be simpler to work with and would not require the creation of an entirely new object in a different class if decompression or splitting are necessary, which would likely be more effective.

It is also possible that a more memory-efficient data structure for the compressed nodes themselves exists. Instead of creating a compressed node class and having each node be an object of that class, it is possible that an alternate data type such as a list or dictionary would consume less memory. This would require a major rewrite of every single function that handles nodes.

5.2 Future Work

There are multiple paths for future research related to $L^{\#}$-radix to take. First and foremost is testing our implementation on large models with hundreds of states to determine whether the memory savings become significant and whether the runtime is negatively affected by a large margin. While our experiments with the manually inserted chain of nodes suggest that this might be the case, it is important to confirm this in a scenario where many operations are performed on the observation tree, instead of simply adding one chain of nodes and measuring the resulting memory consumption. Similarly, running the models m54 and m183 more times than five to test whether the increase in speed is due to a statistical anomaly would be helpful.

We observed that the memory usage was better when $L^{\#}$-radix was run on a chain of a thousand nodes than when it was run on longer chains. It is therefore possible that Python memory allocation makes compressed nodes inefficient when they become too large. We suggest experimenting with capping the size of compressed nodes and seeing whether that uses less memory.

Next, there are multiple ways in which the code of AALpy $L^{\#}$-radix could be improved. It is key to reduce the amount of decompression, as this is likely to cause our implementation of $L^{\#}$-radix to be less memory-efficient than expected. As linked lists are not the most efficient data structure for the storage of the data of compressed nodes, other data structures should also be considered and experimented with.

Lastly, we believe that exploring other possibilities for reducing memory consumption of the $L^{\#}$ algorithm would also be a good direction for future research. There are changes to the algorithm itself that could improve memory usage, such as pruning branches of the observation tree that not currently relevant [7].

Chapter 6

Conclusion

In this thesis, we developed a Python implementation of $L^\#$-radix, a version of $L^\#$ with radix trees, and evaluated its memory consumption when compared to $L^\#$. Our implementation was based on the version of $L^\#$ in the AALpy library, which makes it more accessible.

Our goal was to compare the memory usage of $L^\#$-radix to the original version of the $L^\#$ algorithm in the Python AALpy library. We ran both algorithms on multiple small benchmarking models, all under sixty states, and measured their memory. The results showed either nonexistent or very modest memory savings, suggesting that the models we used for benchmarking were not large enough for compression to be able to reduce memory consumption. This corroborates with the conclusions of Garhewal's work on $L^\#$-radix, where he found that an observation tree had to grow above 500 megabytes for significant results [6]; our models only resulted in observation trees smaller than 170 megabytes.

Further testing with inserting a mock counterexample of a particular length of an observation tree and measuring the difference in memory consumption between $L^\#$ and $L^\#$-radix showed more significant memory savings. $L^\#$-radix proved to be highly effective if the chain of nodes was long enough, using two-thirds less memory than $L^\#$. This strengthens the theory that our benchmarks were too small to see the effect of compression on memory usage. However, further experiments on large models are necessary to be able to confirm this definitively.

Finally, we measured the runtime of $L^\#$-radix and $L^\#$ while running the same benchmarks. We found that the radix tree data structure did not seem to affect the runtime significantly. Nevertheless, our benchmarks were fairly small and likely did not contain much compression. It would therefore be best to run $L^\#$-radix on large models to be able to draw any conclusions about the effectiveness of this algorithm.

Bibliography

[1] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Al Et. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 2013.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987.

[3] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin's Learning. *Electronic Notes in Theoretical Computer Science*, 118:3–18, February 2005.

[4] Frits Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–243, Cham, 2022. Springer International Publishing.

[5] Edi Muškardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering*, 18(3):417–426, September 2022.

[6] Bharat Garhewal. Scalability of Observation Trees in L#. *Unpublished notes*, 2022.

[7] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322, Cham, 2014. Springer International Publishing.

[8] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49, April 2013.

[9] Daniel Neider, Rick Smetsers, Frits Vaandrager, and Harco Kuppens. Benchmarks for Automata Learning and Conformance Testing. In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, pages 390–416. Springer International Publishing, Cham, 2019.

[10] Marc Jasper, Malte Mues, Alnis Murtovi, Maximilian Schlüter, Falk Howar, Bernhard Steffen, Markus Schordan, Dennis Hendriks, Ramon Schiffelers, Harco Kuppens, and Frits W. Vaandrager. RERS 2019: Combining Synthesis with Real-World Models. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 101–115, Cham, 2019. Springer International Publishing.

Appendix A

GitHub Repository

The code written for this thesis can be found at
`https://github.com/k-arolina-t/l-sharp-square-algorithm-radix-trees`.

Appendix B

Tables of Standard Deviations

Model	$L^{\#}$ **standard deviation**	$L^{\#}$**-radix standard deviation**
Gnu-TLS	0.040	0.048
MQTT	0.074	0
Passport	0	0
TCP-Ubuntu	0.185	4.361
Bankcard	0	0.300
m41	0.039	0.621
m54	0.040	0
m183	0	0

Table B.1: Standard deviation of memory usage (in MB) of $L^{\#}$ and $L^{\#}$-radix while running the benchmarks.

Number of Nodes	$L^{\#}$ **standard deviation**	$L^{\#}$**-radix standard deviation**
100	0	0
1 000	0.021	0.001
10 000	0.262	0.173
100 000	0.814	0.305

Table B.2: Standard deviation of memory usage (in MB) of the string of nodes in $L^{\#}$ and $L^{\#}$-radix.

Model	$L^\#$ **standard deviation**	$L^\#$**-radix standard deviation**
Gnu-TLS	0.017	0.012
MQTT	0.004	0.007
Passport	0	0
TCP-Ubuntu	0.145	0.260
Bankcard	0	0
m41	3.539	0.576
m54	0.043	0.013
m183	0.011	0

Table B.3: Standard deviation of runtime (in seconds) of $L^\#$ and $L^\#$-radix while running the benchmarks.