BACHELOR'S THESIS COMPUTING SCIENCE

RADBOUD UNIVERSITY NIJMEGEN

Computing apart states in a partial Mealy machine

Author:
Niek Terol
s1000979

First supervisor/assessor:
dr. J.C. Rot

Second supervisor/assessor:
dr. T.T. Wißmann

Second assessor:
prof. dr. F.W. Vaandrager

July 26, 2022

Acknowledgements

I am very grateful to both of my supervisors, dr. Jurriaan Rot and dr. Thorsten Wißmann, for guiding me through the scientific process of my thesis and for giving invaluable feedback along the way.

Abstract

This thesis presents and analyses an algorithm that finds all pairs of apart states in a partial Mealy machine. While there are already algorithms that accomplish this, they were not analysed extensively. The time complexity of the presented algorithm was found to be $\mathcal{O}(|I|n^2)$, where I is the input alphabet and n the number of states of a given partial Mealy machine. The algorithm was benchmarked to compare the practical performance with the theoretical complexity. The benchmarks show that the time needed for the algorithm to terminate scales quadratically in the number of states. For a fixed number of states, the average performance of the algorithm improves when the number of used input symbols increases.

Contents

Chapter 1

Introduction

Active model learning is the problem of reconstructing a black box finite-state machine by giving it inputs and observing the resulting outputs [7]. Performing this task efficiently is an active field of research in Computer Science. Model learning has numerous applications, such as testing real-world systems. Some of these systems include banking cards, network protocols and old software [7]. The L^* algorithm, created by Angluin [2], is one of the most influential algorithms on active model learning.

L^* makes use of the minimally adequate teacher framework, in which there is a teacher that has full access to the black box finite-state machine. The algorithm can pose membership and equivalence queries to this teacher. A membership query encompasses the algorithm giving an input sequence to the teacher and the teacher giving the output sequence that the black box finite-state machine returned. An equivalence query consists of the algorithm asking the teacher whether its currently constructed finite-state machine is equal to the black box finite-state machine. There are many algorithms in the field of active automata learning that are based on L^* and the minimally adequate teacher framework.

Recently, a new active model learning algorithm called $L^\#$ was presented, which takes a different approach than L^*. The latter reconstructs a finite-state machine based on the equivalence of states, whereas the former does this by using a sort of nonequivalence of states, called apartness [8]. The $L^\#$ algorithm learns a black-box Mealy machine, a type of finite-state machine that outputs a sequence of symbols depending on the sequence of symbols it receives as input. $L^\#$ uses an observation tree, which is a partial Mealy machine in the form of a rooted tree. During the execution of the algorithm, $L^\#$ needs to compute all pairs of states that are apart in the observation tree.

A closely related problem to finding all pairs of states that are nonequivalent is that of the minimisation of deterministic finite automata. Minimising a deterministic finite automaton entails finding a deterministic finite automaton with the least number of states that accepts the same language as the initial automaton. Moore was the first to propose a minimisation algorithm [5], which Hopcroft later improved [3]. These algorithms create a partition that contains sets of equivalent states, resulting in an equivalence relation on the states. Nonequivalence of states is also determined by these algorithms: if two states are not in the same set, then they are nonequivalent. However, the minimisation algorithms of Moore and Hopcroft cannot be used for finding all pairs of apart states in an observation tree, due to its partiality.

While an algorithm that computes all pairs of apart states in a partial Mealy machine in rooted tree form was used to implement $L^{\#}$, it has not been analysed extensively. In this thesis, we present such an algorithm, of which the correctness will be proven and the time complexity analysed. Furthermore, some variations of this algorithm are described that could improve its performance. The algorithm and all of its variations will be benchmarked, in order to compare the performance of the variations and to verify that the theoretical time complexity is accurate.

First, relevant definitions to the problem at hand will be given in chapter 2. Then, the algorithm will be described, its correctness proven and its complexity analysed in chapter 3. Moreover, some variations of the algorithm will be described in chapter 3. The benchmarking process and the results of the benchmarks will be discussed in chapter 4. In chapter 5, related work will be discussed and it will be explained why the minimisation algorithms of Moore and Hopcroft do not function properly on observation trees. Last, in chapter 6 conclusions will be drawn.

Chapter 2

Partial Mealy machines, trees and apartness

In this chapter, concepts will be defined that are relevant to the thesis. Moreover, the mathematical problem that will be solved algorithmically in later chapters is introduced.

Firstly, some preliminary definitions will be given that are relevant to the rest of the thesis in section 2.1. Secondly, partial Mealy machines in rooted tree form will be defined in section 2.2 and partial Mealy trees in section 2.3.

The definition of a partial Mealy machine in rooted tree form will be given as this thesis aims to present an algorithm that finds all pairs of apart states in an observation tree that arises in the active automata learning algorithm $L^{\#}$ [8]. This observation tree is a partial Mealy machine in rooted tree form.

An equivalent representation of partial Mealy machines in rooted tree form, called partial Mealy trees, will also be given in this chapter. Partial Mealy trees are interesting for this thesis, because they lend themselves to be implemented more easily with a functional programming language than partial Mealy machines in rooted tree form. More on this will be discussed in section 2.3 after partial Mealy trees have been defined.

Lastly, it will be shown that partial Mealy machines in rooted tree form are equivalent to partial Mealy trees in section 2.4. As a result of this, an algorithm that finds all pairs of apart states in one of the structures also determines what states are apart in the other structure.

2.1 Preliminary definitions

The partiality in a Mealy machine is represented by partial functions:

Definition 2.1 (Partial function)**.** A *partial function* f is a relation from a set X to a set Y that assigns at most one value $y \in Y$ to each value $x \in X$. A *partial function* f from X to Y is denoted by $f : X \rightharpoonup Y$.

- If $\exists y \in Y : f(x) = y$, we write $f(x)\downarrow$.
- Otherwise, we write $f(x)\uparrow$.

The definition of a rooted tree is needed in order to be able to define when a partial Mealy machine is in rooted tree form.

Definition 2.2 (Rooted tree)**.** A *rooted tree* is a directed graph with a root vertex in which, for every vertex v, there is exactly one path from the root vertex to v.

Some notation will now be introduced that is relevant to Mealy machines. Let X be a set. X^* is defined as the set of all words of $x \in X$ and we write ε for the empty word. Concatenation of two words σ and ρ is written as $\sigma\rho$.

The notion of a prefix-closed set is relevant for the definition of a partial Mealy tree.

Definition 2.3 (Prefix-closed set)**.** Let B be a set. A subset $A \subseteq B^*$ is *prefix-closed* if and only if

$$\forall \sigma \in B^* \quad \forall i \in B : \sigma i \in A \rightarrow \sigma \in A$$

2.2 Partial Mealy machine in rooted tree form

Mealy machines are automata that output a sequence of symbols based on a sequence of input symbols. These machines are the main focus of this thesis.

Definition 2.4 (Mealy machine)**.** A *Mealy machine* is a tuple (I, O, Q, q_0, δ, λ), where

- I is a finite set of input symbols.
- O is a finite set of output symbols.
- Q is a finite set of states.
- $q_0 \in Q$ is an initial state.
- $\delta : Q \times I \rightarrow Q$ is a transition function.
- $\lambda : Q \times I \rightarrow O$ is an output function.

An example Mealy machine can be seen in fig. 2.1. The first symbol in the label of an edge represents the input symbol of that transition and the second symbol represents the output symbol of that transition.

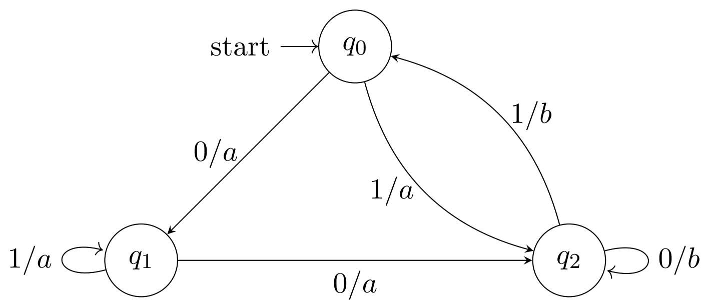

Figure 2.1: Example of a Mealy machine with $I = \{0, 1\}, O = \{a, b\}$

A definition of partial Mealy machines will now be given.

Definition 2.5 (Partial Mealy machine)**.** A *partial Mealy machine* is defined like a Mealy machine, but with a partial transition and output function instead of a total transition and output function:

- $\delta : Q \times I \rightharpoonup Q$
- $\lambda : Q \times I \rightharpoonup O$

Moreover, we require:

$$\forall q \in Q \quad \forall i \in I : \delta(q, i)\downarrow \leftrightarrow \lambda(q, i)\downarrow$$

Note that this definition is more general than a non-partial Mealy machine, since total functions are also partial functions. An example partial Mealy machine can be seen in fig. 2.2.

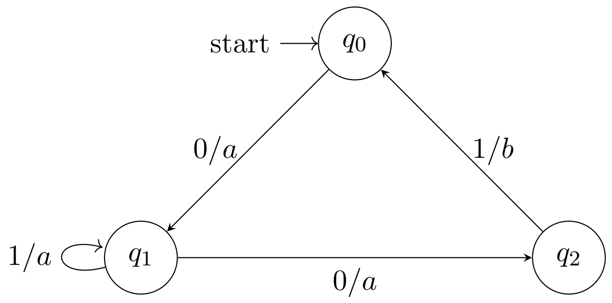

Figure 2.2: Example of a partial Mealy machine with $I = \{0, 1\}, O = \{a, b\}$

Transition functions that take a word as input will now be defined to simplify notation in later definitions, theorems and proofs. This type of transition function is defined recursively using the definition of a standard transition function.

Definition 2.6 (Transition function extended to words)**.** Let $\delta : Q \times I \rightharpoonup Q$ be the transition function of a partial Mealy machine. The *transition function extended to words* $\delta^* : Q \times I^* \rightharpoonup Q$ is defined as follows for $q \in Q, \sigma \in I^*$:

$$\delta^*(q,\sigma) := \begin{cases} q & \text{if } \sigma = \varepsilon \\ \delta^*(\delta(q,i),\sigma') & \text{if } \sigma = i\sigma' \text{ and } \delta(q,i)\downarrow \text{ and } \delta^*(\delta(q,i),\sigma')\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example 2.7. *For the Mealy machine of fig. 2.2, we have:*

$$\begin{aligned} \delta^*(q_0, 010) &= \delta^*(\delta(q_0,0),10) \\ &= \delta^*(q_1,10) \\ &= \delta^*(\delta(q_1,1),0) \\ &= \delta^*(q_1,0) \\ &= \delta^*(\delta(q_1,0),\varepsilon) \\ &= \delta^*(q_2,\varepsilon) \\ &= q_2 \end{aligned}$$

Similar to transition functions, output functions can be defined so that they take a word as input. Output functions that take a word as input will also be recursively defined using the definition of a standard output function.

Definition 2.8 (Output function extended to words)**.** Let $\lambda : Q \times I \rightharpoonup O$ be the output function of a partial Mealy machine. The *output function extended to words* $\lambda^* : Q \times I^* \rightharpoonup O^*$ is defined as follows for $q \in Q, \sigma \in I^*$:

$$\lambda^*(q,\sigma) := \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ \lambda(q,i)\lambda^*(\delta(q,i),\sigma') & \text{if } \sigma = i\sigma' \text{ and } \lambda(q,i)\downarrow \text{ and } \lambda^*(\delta(q,i),\sigma')\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example 2.9. *For the Mealy machine of fig. 2.2, we have:*

$$\begin{aligned} \lambda^*(q_0, 010) &= \lambda(q_0,0)\lambda^*(\delta(q_0,0),10) \\ &= a\lambda^*(q_1,10) \\ &= a\lambda(q_1,1)\lambda^*(\delta(q_1,1),0) \\ &= aa\lambda^*(q_1,0) \\ &= aa\lambda(q_1,0)\lambda^*(q_2,\varepsilon) \\ &= aaa\varepsilon \\ &= aaa \end{aligned}$$

Observation trees are partial Mealy machines that are in the form of a rooted tree. These machines are defined by imposing a requirement on partial Mealy machines.

Definition 2.10 (Partial Mealy machine in rooted tree form)**.** A *partial Mealy machine in rooted tree form* is a partial Mealy machine (I, O, Q, q_0, δ, λ) as in definition 2.5 in which, for every state $q \in Q$, there is exactly one input sequence $\sigma \in I^*$ such that $\delta^*(q_0, \sigma) = q$.

The requirement in definition 2.10 states that, for every state $q \in Q$, there is exactly one path in the partial Mealy machine from the initial state to q. This is in accordance with the definition of a rooted tree as in definition 2.2.

Assumption 2.11. *From now on, whenever a partial Mealy machine is mentioned in this thesis, then it is assumed to be a partial Mealy machine in rooted tree form.*

An example of a partial Mealy machine in rooted tree form can be seen in fig. 2.3. The notation is the same as for Mealy machines.

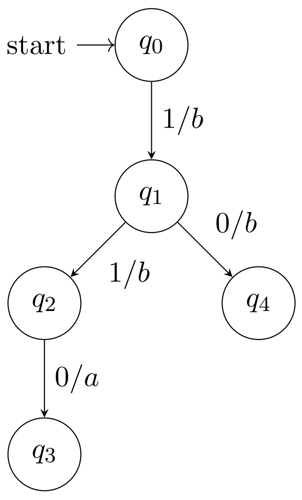

Figure 2.3: Example of a partial Mealy machine in rooted tree form with $I = \{0, 1\}, O = \{a, b\}$

An input sequence that causes a machine to transition from the initial state to a certain other state will be used to show equivalence of partial Mealy machines and partial Mealy trees in section 2.4. Therefore, it is convenient to define this type of input sequence.

Definition 2.12 (Access sequence)**.** Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ be a partial Mealy machine. For each $q \in Q$, an *access sequence* $\text{access}(q) \in I^*$ is a sequence of input symbols such that $\delta^*(q_0, \text{access}(q)) = q$.

Note that for a partial Mealy machine in rooted tree form, there is exactly one access sequence per state. This is because there is exactly one path to each state from the initial state.

Example 2.13. *The access sequence of state q_3 in fig. 2.3 is* 110*.*

Apartness of a pair of states in a partial Mealy machine will now be defined.

Definition 2.14 (Apart states in a partial Mealy machine)**.** Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ be a partial Mealy machine. A pair of states $(p, q) \in Q \times Q$ is *apart* if

$$\exists \sigma \in I^* : \lambda^*(p, \sigma)\downarrow \wedge \lambda^*(q, \sigma)\downarrow \wedge (\lambda^*(p, \sigma) \neq \lambda^*(q, \sigma))$$

The sequence of input symbols σ is called a distinguishing sequence of p and q. The fact that the states p and q are apart is denoted by $p \# q$.

Example 2.15. *The states q_1 and q_2 in the machine in fig. 2.3 are apart, as witnessed by the distinguishing sequence* 0*.*

The states q_0 and q_1 in the machine in fig. 2.3 are also apart, as witnessed by the distinguishing sequence is 10*.*

Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ be a partial Mealy machine. The problem this thesis tries to solve algorithmically is to find all pairs of states $(p, q) \in Q \times Q$ that are apart in $\mathcal{M}$.

2.3 Partial Mealy tree

In this section, a more compact representation of tree-shaped Mealy machines, called partial Mealy trees, is provided.

Definition 2.16 (Partial Mealy tree)**.** A *partial Mealy tree* is a tuple (I, O, Q, λ), where:

- I is a finite set of input symbols.
- O is a finite set of output symbols.
- $Q \subset I^*$ is a finite non-empty prefix-closed set of states. The initial state is ε. The state $qi \in Q$ is reached by giving i as input symbol when the partial Mealy tree is in state q.
- $\lambda : Q \times I \rightharpoonup O$ is an output function.
- $\lambda(q, i)\downarrow \leftrightarrow qi \in Q$.

The prefix order on Q causes a partial Mealy tree to naturally induce a rooted tree.

A partial Mealy tree can be implemented more easily using a functional programming language than a partial Mealy machine because the states are not defined explicitly. Moreover, only an output function has to be implemented instead of both a transition and an output function.

An example partial Mealy tree can be seen in fig. 2.4. The name of a state represents the sequence of input symbols needed to transition from ε to that state. This means that state $\sigma\rho$ can be reached from the state σ by giving ρ as input, where $\sigma, \rho \in I^*$. The label on an edge represents the output symbol associated with that transition.

For a fixed sequence of input symbols, the partial Mealy tree in fig. 2.4 outputs the same sequence of symbols as the partial Mealy machine in fig. 2.3. The difference between the two automata is that the one in fig. 2.4 is represented as a partial Mealy tree and the one in fig. 2.3 is represented as a partial Mealy machine.

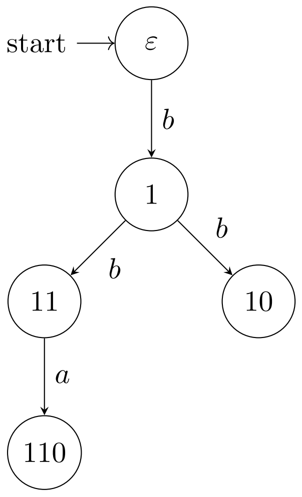

Figure 2.4: Example of a partial Mealy tree with $I = \{0, 1\}, O = \{a, b\}$

The definition of output functions that take a word as input will now be given for partial Mealy trees.

Definition 2.17 (Output function extended to words)**.** Let $\lambda : Q \times I \rightharpoonup O$ be the output function of a partial Mealy tree. The *output function extended to words* $\lambda^* : Q \times I^* \rightharpoonup O^*$ is defined as follows for $q \in Q, \sigma \in I^*$:

$$\lambda^*(q, \sigma) := \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ \lambda(q, i)\lambda^*(qi, \sigma') & \text{if } \sigma = i\sigma' \text{ and } qi \in Q \text{ and } \lambda^*(qi, \sigma')\downarrow \\ \text{undefined} & \text{otherwise} \end{cases}$$

Example 2.18. *In fig. 2.4, we have*

$$\begin{aligned}\lambda^*(\varepsilon, 10) &= \lambda(\varepsilon, 1)\lambda^*(1, 0)\\ &= b\lambda^*(1, 0)\\ &= b\lambda(1, 0)\lambda^*(10, \varepsilon)\\ &= bb\varepsilon\\ &= bb\end{aligned}$$

Apartness of a pair of states in a partial Mealy tree will now be defined.

Definition 2.19 (Apart states in a partial Mealy tree)**.** Let $\mathcal{T} = (I, O, Q, \lambda)$ be a partial Mealy tree. A pair of states $(p, q) \in Q \times Q$ is apart if

$$\exists \sigma \in I^* : p\sigma \in Q \wedge q\sigma \in Q \wedge (\lambda^*(p, \sigma) \neq \lambda^*(q, \sigma))$$

The sequence of input symbols σ is called a distinguishing sequence of p and q. The fact that the states p and q are apart is denoted by $p \# q$.

Example 2.20. *The states* 1 *and* 11 *are apart in fig. 2.4, as witnessed by the distinguishing sequence* 0*. The states* ε *and* 1 *are apart as well in fig. 2.4, as witnessed by the distinguishing sequence* 10*.*

Let $\mathcal{T} = (I, O, Q, \lambda)$ be a partial Mealy tree. The problem that this thesis tries to solve algorithmically is to find all pairs of states $(p, q) \in Q \times Q$ that are apart in $\mathcal{T}$.

2.4 Equivalence of definitions

In this section, the equivalence of partial Mealy machines and partial Mealy trees will be shown. The following two statements will be proven in order to show that partial Mealy machines and partial Mealy trees are equivalent with respect to apart states.

1. For each partial Mealy machine $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ there is a partial Mealy tree $\mathcal{T} = (I', O', Q', \lambda')$ such that if the pair of states $(p, q) \in Q \times Q$ is apart in $\mathcal{M}$, then $(\text{access}(p), \text{access}(q)) \in Q' \times Q'$ is apart in $\mathcal{T}$.

2. For each partial Mealy tree $\mathcal{T} = (I', O', Q', \lambda')$ there is a partial Mealy machine $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ such that if the pair of states $(p, q) \in Q' \times Q'$ is apart in $\mathcal{T}$, then $(\delta^*(q_0, p), \delta^*(q_0, q)) \in Q \times Q$ is apart in $\mathcal{M}$.

These statements will be proven using two constructions. The first construction transforms a partial Mealy machine into a partial Mealy tree and the second construction transforms a partial Mealy tree into a partial Mealy machine. The first construction will give the desired partial Mealy tree of the first statement and the second construction the desired partial Mealy machine of the second statement.

2.4.1 Proof of first statement

In this section, the first statement of section 2.4 will be proven. A construction will first be defined that transforms a given partial Mealy machine into a partial Mealy tree. This construction will then be used to prove the first statement.

Construction 2.21. *Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ be a partial Mealy machine. A partial Mealy tree $\mathcal{T} = (I', O', Q', \lambda')$ is constructed from $\mathcal{M}$ as follows:*

- $I' := I$
- $O' := O$
- $Q' := \{\sigma \in I^* \mid \delta^*(q_0, \sigma)\downarrow\}$
- $\lambda'(q', i) := \begin{cases} \lambda(\delta^*(q_0, q'), i) & \textit{if } \lambda(\delta^*(q_0, q'), i)\downarrow \\ \textit{undefined} & \textit{otherwise} \end{cases}$

Note that since $\mathcal{M}$ is a partial Mealy machine in rooted tree form, Q is a finite set. Therefore, the construction is well-defined.

Example 2.22. *Let the partial Mealy machine depicted in fig. 2.3 be a partial Mealy machine $\mathcal{M}$. The partial Mealy tree $\mathcal{T}$ that is the result of construction 2.21 when applied to $\mathcal{M}$ is illustrated in fig. 2.4.*

Definition 2.16 requires that a partial Mealy tree $\mathcal{T} = (I', O', Q', \lambda')$ satisfies three conditions.

Lemma 2.23. *A partial Mealy tree that is a result of construction 2.21 satisfies the three conditions of definition 2.16:*

- *Q' is non-empty.*
- *Q' is prefix-closed.*
- $\forall q \in Q' \quad \forall i \in I' : \lambda'(q, i)\downarrow \leftrightarrow qi \in Q'$

Proof. The proof will be divided into three parts, one for each condition. Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ be a partial Mealy machine. Let $\mathcal{T} = (I', O', Q', \lambda')$ be the partial Mealy tree that is a result of applying construction 2.21 to $\mathcal{M}$.

- First, it will be proven that Q' is non-empty. We know that $\delta^*(q_0, \varepsilon)\downarrow$. From the construction we can see that $\varepsilon \in Q'$. Thus, Q' is non-empty.
- Second, it will be proven that Q' is prefix-closed. Assume that Q' is not prefix-closed. Then there exist $\sigma, \rho \in I^*$ such that $\sigma\rho \in Q' \wedge \sigma \notin Q'$. From this follows $\delta^*(q_0, \sigma\rho)\downarrow$ and $\delta^*(q_0, \sigma)\uparrow$. From definition 2.6 follows $\delta^*(q_0, \sigma\rho)\uparrow$. This is a contradiction, so Q' is prefix-closed.

- Last, it will be proven that:

  $$\forall q \in Q' \quad \forall i \in I' : \lambda'(q,i){\downarrow} \leftrightarrow qi \in Q'$$

  Let $q \in Q'$ and $i \in I'$.

  Assume $\lambda'(q,i){\downarrow}$. Then we have $\lambda(\delta^*(q_0,q),i){\downarrow}$ because of construction 2.21. From definition 2.5 follows that $\delta(\delta^*(q_0,q),i){\downarrow}$. Now it follows from definition 2.6 that $\delta^*(q_0,qi){\downarrow}$. Construction 2.21 now gives us $qi \in Q'$. Thus, if $\lambda'(q,i){\downarrow}$, then $qi \in Q'$.

  Now assume $qi \in Q'$. From construction 2.21 follows $\delta^*(q_0,qi){\downarrow}$. Now it follows from definition 2.6 that $\delta(\delta^*(q_0,q),i){\downarrow}$. From definition 2.5 it follows that $\lambda(\delta^*(q_0,q),i){\downarrow}$. From construction 2.21 we now have $\lambda'(q,i){\downarrow}$. Thus, if $qi \in Q'$, then $\lambda'(q,i){\downarrow}$.

  Since $q \in Q'$ and $i \in I'$ were chosen arbitrarily, this holds for all $q \in Q'$ and $i \in I'$.

Thus, a partial Mealy tree resulting from construction 2.21 satisfies the three conditions. □

Now it will proven that construction 2.21 reflects the apartness relation.

Theorem 2.24. *For each partial Mealy machine $\mathcal{M} = (I,O,Q,q_0,\delta,\lambda)$, we have that if the pair of states $(p,q) \in Q \times Q$ is apart in $\mathcal{M}$, then the pair of states $(access(p), access(q)) \in Q' \times Q'$ is apart in the partial Mealy tree $\mathcal{T} = (I',O',Q',\lambda')$ that is a result of construction 2.21 applied to $\mathcal{M}$.*

Proof. Let $\mathcal{M} = (I,O,Q,q_0,\delta,\lambda)$ be a partial Mealy machine and let $\mathcal{T} = (I',O',Q',\lambda')$ be the partial Mealy tree that is the result of applying construction 2.21 to $\mathcal{M}$. Assume that the pair of states $(p,q) \in Q \times Q$ is apart in $\mathcal{M}$. Then there is an input sequence $\sigma \in I^*$, such that $\lambda^*(p,\sigma){\downarrow}, \lambda^*(q,\sigma){\downarrow}$ and $\lambda^*(p,\sigma) \neq \lambda^*(q,\sigma)$. From the definition of partial Mealy machines follows that $\delta^*(p,\sigma){\downarrow}$ and $\delta^*(q,\sigma){\downarrow}$. Since $\delta^*(p,\sigma) = \delta^*(q_0,\text{access}(p)\sigma)$ and $\delta^*(q,\sigma) = \delta^*(q_0,\text{access}(q)\sigma)$, we have $\delta^*(q_0,\text{access}(p)\sigma){\downarrow}$ and $\delta^*(q_0,\text{access}(q)\sigma)\,{\downarrow}$. Now we know that $\text{access}(p)\sigma \in Q'$, $\text{access}(q)\sigma \in Q'$. Moreover, we have

$$\begin{aligned}
&\lambda^{*\prime}(\text{access}(p),\sigma)\\
&= \lambda^*(\delta^*(q_0,\text{access}(p)),\sigma)\\
&= \lambda^*(p,\sigma)\\
&\neq \lambda^*(q,\sigma)\\
&= \lambda^*(\delta^*(q_0,\text{access}(q)),\sigma)\\
&= \lambda^{*\prime}(\text{access}(q),\sigma)
\end{aligned}$$

Thus, the pair $(\text{access}(p),\text{access}(q)) \in Q' \times Q'$ is apart in $\mathcal{T}$. Since $\mathcal{M}$ was chosen arbitrarily, this proof holds for all partial Mealy machines. □

2.4.2 Proof of second statement

In this section, the second statement of section 2.4 will be proven. A construction will first be defined that transforms a given partial Mealy tree into a partial Mealy machine. This construction will then be used to prove the second statement.

Construction 2.25. *Let $\mathcal{T} = (I', O', Q', \lambda')$ be a partial Mealy tree. A partial Mealy machine $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ is constructed from $\mathcal{T}$ as follows:*

- $I := I'$
- $O := O'$
- $Q := Q'$
- $q_0 := \varepsilon$
- $\delta(q, i) := \begin{cases} qi & \textit{if } qi \in Q \\ \textit{undefined} & \textit{otherwise} \end{cases}$
- $\lambda(q, i) := \begin{cases} \lambda'(q, i) & \textit{if } \lambda'(q, i)\downarrow \\ \textit{undefined} & \textit{otherwise} \end{cases}$

Note that $\mathcal{T}$ is a partial Mealy tree, so Q is a finite set. Therefore, the construction is well-defined.

Example 2.26. *The partial Mealy machine depicted in fig. 2.5 is the result of applying construction 2.25 to the partial Mealy tree in fig. 2.4.*

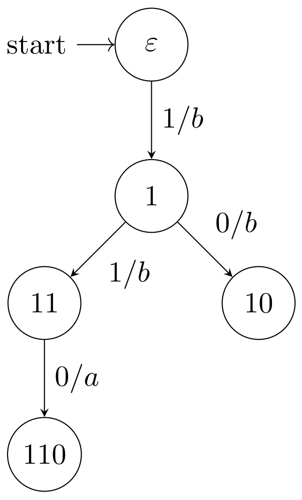

Figure 2.5: Example of a partial Mealy machine that is a result of construction 2.25 ($I = \{0, 1\}, O = \{a, b\}$)

From definition 2.5 it can be seen that a partial Mealy machine $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ must satisfy the following condition:

$$\forall q \in Q \quad \forall i \in I : \delta(q,i)\downarrow \leftrightarrow \lambda(q,i)\downarrow$$

It will first be proven that a partial Mealy machine that is a result of construction 2.25 satisfies this condition.

Lemma 2.27. *For each partial Mealy machine $\mathcal{M}$ = (I, O, Q, q_0, δ, λ) that is a result of construction 2.25, we have*

$$\forall q \in Q \quad \forall i \in I : \delta(q,i)\downarrow \leftrightarrow \lambda(q,i)\downarrow$$

Proof. Let $\mathcal{T} = (I', O', Q', \lambda')$ be a partial Mealy tree and let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ be the partial Mealy machine that is the result of applying construction 2.25 to $\mathcal{T}$. Let $q \in Q$ and $i \in I$.

Assume that $\delta(q,i)\downarrow$. Then we know from construction 2.25 that $qi \in Q$. From definition 2.16 we know $\lambda'(q,i)\downarrow$. From construction 2.25 now follows $\lambda(q,i)\downarrow$. Thus, if $\delta(q,i)\downarrow$, then $\lambda(q,i)\downarrow$.

Now assume that $\lambda(q,i)\downarrow$. Then we know from construction 2.25 that $\lambda'(q,i)\downarrow$. From definition 2.16 we know that $qi \in Q$. From construction 2.25 we now have $\delta(q,i)\downarrow$. Thus, if $\lambda(q,i)\downarrow$, then $\delta(q,i)\downarrow$.

Since $q \in Q$ and $i \in I$ were chosen arbitrarily, this holds for all $q \in Q$ and $i \in I$. □

Now it will be proven that construction 2.25 reflects the apartness relation.

Theorem 2.28. *For each partial Mealy tree $\mathcal{T} = (I', O', Q', \lambda')$, we have that if the pair of states $(p, q) \in Q' \times Q'$ is apart in $\mathcal{T}$, then the pair of states $(p, q) \in Q \times Q$ is apart in the partial Mealy machine $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ that is a result of construction 2.25 applied to $\mathcal{T}$.*

Proof. Let $\mathcal{T} = (I', O', Q', \lambda')$ be a partial Mealy tree and let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$ be a partial Mealy machine that is the result of applying construction 2.25 to $\mathcal{T}$. Assume that the pair of states $(p, q) \in Q' \times Q'$ is apart in $\mathcal{T}$. Then we know that there is a $\sigma \in I'$ such that $p\sigma \in Q', q\sigma \in Q'$ and $\lambda^{*\prime}(p,\sigma) \neq \lambda^{*\prime}(q,\sigma)$. From definition 2.16 follows that $\lambda^{*\prime}(p,\sigma)\downarrow$ and $\lambda^{*\prime}(q,\sigma)\downarrow$. From the definition of λ in construction 2.25 follows that $\lambda^*(p,\sigma)\downarrow, \lambda^*(q,\sigma)\downarrow$ and $\lambda^*(p,\sigma) \neq \lambda^*(q,\sigma)$. Thus, the pair of states $(p,q) \in Q \times Q$ is apart in $\mathcal{M}$. Since $\mathcal{T}$ was chosen arbitrarily, this proof holds for all partial Mealy trees. □

To conclude, partial Mealy machines and partial Mealy trees are equivalent in terms of apartness.

Chapter 3

Algorithmic computation of apart states

In this chapter, an algorithm for computing all pairs of states that are apart in a partial Mealy machine will be described. This was created for partial Mealy machines, as this structure is more straightforward to implement with an imperative programming language. However, it could be modified to be applicable to partial Mealy trees. The algorithm consists of two smaller ones, called DetermineApart and ComputeAllApartPairs. The first determines whether a given pair of states is apart. The second determines for all pairs of states in a given partial Mealy machine whether they are apart. ComputeAllApartPairs accomplishes this by calling DetermineApart for each pair of states in the partial Mealy machine.

3.1 Description of DetermineApart

DetermineApart computes whether a pair of states (p, q) is apart. This is done by checking if p and q output a different symbol when they receive the same input symbol. If p and q do not output different symbols, then the algorithm checks recursively for all input symbols i whether the children $\delta(p, i)$ and $\delta(q, i)$ are apart. If they are apart with distinguishing sequence σ, then p and q are apart with distinguishing sequence $i\sigma$. This fact will be used in the correctness proof but DetermineApart does not return distinguishing sequences. There is a small description at the end of this section that explains how DetermineApart can be modified to return distinguishing sequences. After the above options have been explored, p and q are found not to be apart.

DetermineApart is a recursive algorithm that utilises dynamic programming and a map from state pairs to booleans. When a pair of states is found to be either apart or not apart, this is immediately stored in the map. DetermineApart checks this map before computing whether a pair of states is apart. The algorithm becomes more efficient using this map, because a large part of it is not executed if the given pair of states is already in the map. After the algorithm has terminated, all pairs that are apart have the value `true` in the map. Conversely, all pairs that are not apart have the value `false`.

The pairs in the map are represented as unordered pairs. This cuts the memory needed for the map in half. Representing the pairs as unordered pairs is achieved by imposing an ordering on the states. A pair (p, q) will only be stored in the map if p is smaller than q in the ordering.

The symbols used in DetermineApart are similar to the symbols used to describe Mealy machines:

- A is a map from state pairs to booleans. The map is passed by reference.
- p and q are states.
- I is an input alphabet.
- δ is a transition function, implemented as a map.
- λ is an output function, implemented as a map.

Algorithm 1 DetermineApart($A, p, q, I, \delta, \lambda$)

```
if $p > q$ then
    return DetermineApart($A, q, p, I, \delta, \lambda$)
end if
if $A[(p, q)] \neq$ NIL then
    return $A[(p, q)]$
end if
for all $i \in I$ with $\lambda(p, i)\downarrow$ and $\lambda(q, i)\downarrow$ do
    if $\lambda(p, i) \neq \lambda(q, i)$ then
        $A[(p, q)] \leftarrow$ true
        return true
    end if
end for
for all $i \in I$ with $\delta(p, i)\downarrow$ and $\delta(q, i)\downarrow$ do
    if DetermineApart($A, \delta(p, i), \delta(q, i), I, \delta, \lambda$) = true then
        $A[(p, q)] \leftarrow$ true
        return true
    end if
end for
$A[(p, q)] \leftarrow$ false
return false
```

At lines 1-3, the algorithm checks whether p is larger in the ordering than q. If this is the case, then the algorithm will be called on the pair (q, p). This is done to make sure that A remains a map of unordered pairs. At lines 4-6, the algorithm checks whether the pair (p, q) is already in the map A. If this is the case, then the algorithm returns the value $A[(p, q)]$ and terminates. At lines 7-12, the algorithm checks whether the given states p and q output different symbols when they receive the same input symbol. At lines 13-18, the algorithm checks recursively, for all input symbols $i \in I$, whether $\delta(p, i)$ and $\delta(q, i)$ are apart. If there is a $i \in I$ such that they are apart, then p and q are also apart. At lines 19-20, the algorithm knows that the given pair of states is not apart and it puts this pair of states in A with the value `false`.

In order to make DetermineApart compatible with partial Mealy trees instead of partial Mealy machines, the following changes would have to be made:

- The transition function δ has to be changed to a set of states Q.
- The clauses at lines 7 and 13 have to be changed from

  $$i \in I \text{ with } \lambda(p, i){\downarrow} \text{ and } \lambda(q, i){\downarrow}$$

  to

  $$i \in I \text{ with } pi \in Q \text{ and } qi \in Q$$

Note that DetermineApart could be adjusted to return a distinguishing sequence for each pair of apart states. This can be achieved by changing A to a map from state pairs to input symbols and by using the following conventions:

- If $A[(p, q)] = \text{NIL}$, then it has not been determined yet whether p and q are apart or not.
- If $A[(p, q)] = \varepsilon$, then p and q are not apart.
- If $A[(p, q)] = \sigma$ with $\sigma \in I^*$, then p and q are apart and a distinguishing sequence for this pair of states is σ.

3.2 Description of ComputeAllApartPairs

ComputeAllApartPairs ensures that every pair of states is present in the map A.

Algorithm 2 ComputeAllApartPairs(I, Q, δ, λ)

```
1: $A$ is an empty map from state pairs to booleans
2: for all $p, q \in Q, p < q$ do
3:     DetermineApart($A, p, q, I, \delta, \lambda$)
4: end for
5: return $A$
```

3.3 Correctness

The correctness of both algorithms will be shown in this section. This is done by proving the correctness of DetermineApart using an invariant on the map A, which is defined as follows:

$$\text{Inv}(A) = \forall p, q \in Q \text{ with } (p, q) \in A : A[(p, q)] = \text{true} \iff p \# q$$

ComputeAllApartPairs is correct if A is defined for every unordered pair of states after it has finished. Since no values are removed from A in ComputeAllApartPairs and DetermineApart, the correctness of the former follows trivially from the correctness of the latter. Hence, only correctness of DetermineApart will be proven, which is represented by the following theorem.

Theorem 3.1. *If Inv(A) holds before DetermineApart(A, p, q, I, δ, λ) is called, then, after DetermineApart(A, p, q, I, δ, λ) has finished, we have:*

- $(p, q) \in A$ *or* $(q, p) \in A$
- *Inv(A) still holds*

Proof. For the entire proof, assume that Inv(A) holds before Determine-Apart is called.

Theorem 3.1 will be proven by induction over the number of recursive calls. DetermineApart always terminates, because it recurses over a finite tree. Therefore, a proof by induction can be used.

DetermineApart can only terminate at line 2, 5, 10, 16 or 20. Since the algorithm always terminates, we know that one of these lines will be reached during its execution.

If line 2 is reached, then A has not been modified by the algorithm before line 2. Consequently, Inv(A) holds before the recursive call at line 2. Since Inv(A) holds before the recursive call, we have after the recursive call:

- $(p, q) \in A$ or $(q, p) \in A$
- Inv(A) holds

The algorithm terminates immediately after the recursive call. Because it terminates, DetermineApart is correct if it reaches line 2.

If line 5 is reached, then A has not been modified before line 5. Therefore, Inv(A) still holds. We also know that $(p, q) \in A$. The algorithm terminates at line 5. Because it terminates, DetermineApart is correct if it reaches line 5.

If line 10 is reached, then A has not been altered before line 9 and thus Inv(A) holds. We know that there is a $i \in I$ such that $\lambda(p, i)\downarrow$, $\lambda(q, i)\downarrow$ and $\lambda(p, i) \neq \lambda(q, i)$. As a result we have $p \# q$ with distinguishing sequence i. At line 9, the pair (p, q) is added to A and this pair is assigned the value `true`. This operation does not violate Inv(A), so the invariant holds for the updated map A. The algorithm terminates at line 10. Because it terminates, DetermineApart is correct if it reaches line 10.

If line 16 is reached, then line 14 has been executed. We know that if line 14 is reached, then A has not been altered before line 14. Therefore, Inv(A) holds before the recursive call. Since Inv(A) holds before the recursive call, we have after the recursive call:

- $(\delta(p, i), \delta(q, i)) \in A$ or $(\delta(q, i), \delta(p, i)) \in A$ for some $i \in I$
- Inv(A) holds

We also know that if line 16 is reached, then there is an $i \in I$ such that $\delta(p, i) \# \delta(q, i)$. Let a distinguishing sequence of $\delta(p, i)$ and $\delta(q, i)$ be $\sigma \in I^*$. Then the sequence $i\sigma$ is a distinguishing sequence of p and q. Hence, p and q are apart. At line 15, the pair (p, q) is added to A and this pair is assigned the value `true`. This operation does not violate Inv(A), so the invariant holds for the updated map A. The algorithm terminates at line 16. Because it terminates, DetermineApart is correct if it reaches line 16.

If line 20 is reached, then A could have been modified at line 14. However, it was shown that Inv(A) holds after the recursive call at line 14. We know that there is no distinguishing sequence $i\sigma \in I^*$, with $i \in I, \sigma \in I^*$, of p and q. If there would be a distinguishing sequence $i\sigma$, then the algorithm would reach either line 10 or line 16. In case σ is the empty word, line 10 is reached. In case σ is not the empty word, line 16 is reached. Consequently, we know that p and q are not apart. The algorithm adds (p, q) to A and this pair is assigned the value `false`. This operation does not violate Inv(A), so the invariant holds for the updated map A. The algorithm terminates at line 20. Because it terminates, DetermineApart is correct if it reaches line 20. $\square$

3.4 Complexity

In this section the time complexity of DetermineApart and of ComputeAll-ApartPairs will be analysed. For the remainder of this section, assume that

A is implemented as a hash map and that a suitable hash function is used, so that there are few collisions. Searching for an element and insertion of an element in the hash map A both have an amortized time complexity of $\mathcal{O}(1)$.

3.4.1 Complexity of DetermineApart

DetermineApart(A, p, q, I, δ, λ) is called on partial Mealy machines that are trees and the recursive calls at line 14 are called on the children of the current states. As a result of this, every state $v \in Q$ can appear at most once as parameter p in the recursive calls. Thus, there are at most $|Q|$ recursive calls. Since each given partial Mealy machine is a tree, it will have at most $|Q| - 1$ transitions. The for-loops at lines 7-12 and lines 13-18 are both executed at most once for each transition, so these for-loops add $\mathcal{O}(|Q|)$ to the total time complexity. Hence, the time complexity of DetermineApart is $\mathcal{O}(|Q|)$.

3.4.2 Complexity of ComputeAllApartPairs

Let n be the number of states in a given partial Mealy tree. Compute-AllApartPairs makes calls to DetermineApart and DetermineApart makes recursive calls to itself. There are two cases if DetermineApart(A, p, q, I, δ, λ) is called:

1. (p, q) or (q, p) is in A.
2. (p, q) and (q, p) are not in A.

In case 2, at most $\mathcal{O}(|I|)$ steps are taken, without considering the steps in the recursive calls. DetermineApart makes at most $|I|$ recursive calls at line 14. The pair (p, q) cannot appear again in recursive calls, since DetermineApart is called on the children of p and q and the partial Mealy machine is a tree. When DetermineApart(A, p, q, I, δ, λ) terminates, it is guaranteed that (p, q) or (q, p) is in A. Thus, case 2 happens at most once per pair of states. Since there are n^2 pairs of states, DetermineApart can be in case 2 at most n^2 times. This gives a total complexity of $\mathcal{O}(|I|n^2)$ for the calls that end up in case 2.

DetermineApart is in case 1 for all other calls. In case 1, we have that DetermineApart(A, p, q, I, δ, λ) either terminates at line 2 or at line 5. If it terminates at line 2, then DetermineApart(A, q, p, I, δ, λ) is called recursively and this recursive call terminates at line 5, taking $\mathcal{O}(1)$ time. If DetermineApart(A, p, q, I, δ, λ) terminates at line 5, then it also takes $\mathcal{O}(1)$ time. Thus, case 1 always takes $\mathcal{O}(1)$ time. Every time the algorithm is in case 2, it is possible that a following recursive call ends up in case 1. DetermineApart is in case 2 at most n^2 times and each time it is in case

2, it can make at most $|I|$ recursive calls. As a result, there are at most $|I|n^2$ recursive calls possible that end up in case 1. ComputeAllApartPairs calls DetermineApart for each pair of states, so n^2 times in total. Therefore, DetermineApart is at most $|I|n^2 + n^2$ times in case 1, taking $\mathcal{O}(|I|n^2 + n^2)$ time. The resulting time complexity of ComputeAllApartPairs is $\mathcal{O}(|I|n^2 + |I|n^2 + n^2) = \mathcal{O}(|I|n^2)$.

3.5 Variations

There are some variations of the algorithm that could have an effect on its efficiency. These variations will be described in this section.

3.5.1 State order in ComputeAllApartPairs

The order in which states are chosen at line 2 of ComputeAllApartPairs could influence the efficiency of the algorithm. The number of times that DetermineApart is called in total depends on the order in which the states are chosen. This is because recursive calls can render calls from Compute-AllApartPairs to DetermineApart redundant, as a pair of states might have been put in the map A by a recursive call. However, there is an option for which no redundant calls from ComputeAllApartPairs will be made.

For the sake of simplicity, assume that if a state p is a parent of a state q, then $p < q$. This implies that the root state is smaller than every other state. Three possible orders of states are:

1. ComputeAllApartPairs picks pairs of states randomly. This is the default of ComputeAllApartPairs. This option acts as a benchmark for the other two options. This option is called ‘random’.

2. ComputeAllApartPairs starts at the top of the tree structure and works its way down. This option is called ‘top-down’. The order of the pairs is determined as in algorithm 3.

3. ComputeAllApartPairs starts with leaf states of the tree structure and works its way up. This option is called ‘bottom-up’. Inverting the order of the pairs in ‘top-down’ results in the order of the pairs in ‘bottom-up’.

Algorithm 3 TopDownOrder(Q)

```
1: $L$ is an empty array of pairs of states.
2: for all $p \in Q$, from lowest to highest state in the order do
3:     for all $q \in Q, p < q$, from lowest to highest state in the order do
4:         $L \leftarrow (p, q)$
5:     end for
6: end for
7: return $L$
```

The order in which the states are called in ComputeAllApartPairs is not relevant for the correctness proof in section 3.3 and therefore DetermineApart is still correct. ComputeAllApartPairs ensures that A is defined for every unordered pair of states using DetermineApart, regardless of the order in which those pairs are chosen. Hence, correctness of ComputeAllApartPairs follows trivially from the correctness of DetermineApart.

The time complexity of ComputeAllApartPairs does not depend on the states p and q that it receives as input. From the complexity analysis in section 3.4.2 follows that DetermineApart ends up in case 2 at most once for each unordered pair of states, taking $\mathcal{O}(|I|n^2)$ time in total. The order in which the states are chosen in the for-loop of ComputeAllApartPairs does not change this fact. The number of times that DetermineApart ends up in case 1 depends on the option of this variation that is used, but it never exceeds $|I|n^2 + n^2$. Thus, the complexity of ComputeAllApartPairs remains $\mathcal{O}(|I|n^2)$ for the options of this variation.

However, the performance of ComputeAllApartPairs in practice could be affected by the order in which states are picked. In case ComputeAllApartPairs starts at leaf states and works its way up, then every recursive call will terminate at line 5. This means that the algorithm is in case 2 of the complexity analysis in section 3.4.2 if and only if DetermineApart was called directly from ComputeAllApartPairs. In the complexity analysis it was concluded that ComputeAllApartPairs takes $\mathcal{O}(|I|n^2 + |I|n^2 + n^2)$ steps in total. This would now be reduced to $\mathcal{O}(|I|n^2 + |I|n^2)$, as the term n^2 represents the calls to DetermineApart from ComputeAllApartPairs that end up in case 1. These calls from ComputeAllApartPairs to DetermineApart are redundant, as they do not add information to the map A. If the algorithm starts at the root state and works its way down the tree structure or when states are picked randomly, then there will be redundant calls from ComputeAllApartPairs to DetermineApart. Consequently, it is expected that the 'bottom-up' option gives the best performance. This is in accordance with the general rule that bottom-up causes less overhead than top-down in recursive algorithms.

3.5.2 Order of for-loops in DetermineApart

A second possible variation, called DetermineApartReversed, consists of changing the order of the for-loops of DetermineApart. The for-loop at lines 13-18 is placed in front of the one at lines 7-12, as shown in the pseudocode of algorithm 4. The result of this is that DetermineApartReversed first checks whether children of the given states are apart. DetermineApartReversed could be faster than DetermineApart, provided that there are already numerous pairs of apart states in the map A. This is because some runs of DetermineApartReversed will then terminate before its second for-loop.

Algorithm 4 DetermineApartReversed($A, p, q, I, \delta, \lambda$)

```
 1: if $p > q$ then
 2:     return DetermineApart($A, q, p, I, \delta, \lambda$)
 3: end if
 4: if $A[(p, q)] \neq$ NIL then
 5:     return $A[(p, q)]$
 6: end if
 7: for all $i \in I$ with $\delta(p, i)\downarrow$ and $\delta(q, i)\downarrow$ do
 8:     if DetermineApart($A, \delta(p, i), \delta(q, i), I, \delta, \lambda$) = true then
 9:         $A[(p, q)] \leftarrow$ true
10:         return true
11:     end if
12: end for
13: for all $i \in I$ with $\lambda(p, i)\downarrow$ and $\lambda(q, i)\downarrow$ do
14:     if $\lambda(p, i) \neq \lambda(q, i)$ then
15:         $A[(p, q)] \leftarrow$ true
16:         return true
17:     end if
18: end for
19: $A[(p, q)] \leftarrow$ false
20: return false
```

DetermineApartReversed is correct, as the correctness of Determine-Apart does not depend on the order of the for-loops. The correctness of the standard version does rely on the fact that both for loops are executed before line 19 is reached, but this still happens in DetermineApartReversed. Correctness of ComputeAllApartPairs follows trivially from the correctness of DetermineApartReversed.

The fact that the for-loops are rearranged does not influence the time complexity of ComputeAllApartPairs. DetermineApartReversed still ends up at most n^2 times in case 2 of the complexity analysis in section 3.4.2 and at most $|I|n^2 + n^2$ times in case 1. However, the practical efficiency of ComputeAllApartPairs could be influenced by this variation. There are two cases:

- A does not contain a great number of pairs of apart states. DetermineApartReversed makes more recursive calls than DetermineApart, as it first makes recursive calls and only then proceeds to the for-loop at lines 13-18 of algorithm 4. As a result of the increase in the number of recursive calls, ComputeAllApartPairs makes more calls to DetermineApartReversed that end up in case 1 of the complexity analysis. These are redundant calls, as was stated in section 3.5.1, which is why it is expected that the performance is worse in this case.

- A already contains a great number of pairs of apart states. The argument of the previous bullet point still holds in this case. However, it becomes increasingly likely that a recursive call at line 8 of DetermineApartReversed returns true when there are more pairs of apart states in the map A. As a consequence, some runs of DetermineApartReversed take fewer steps, because the for-loop at lines 13-18 of DetermineApartReversed does not have to be executed if a recursive call at line 8 returns true. This might make up for the loss of performance described in the first bullet point.

Chapter 4

Benchmark

The algorithm ComputeAllApartPairs was benchmarked to check whether the time complexity is accurate and to compare the variations described in section 3.5. First, an algorithm for generating random Mealy machines will be presented in section 4.1. Then, the effect that properties of partial Mealy machines have on the performance of ComputeAllApartPairs will be discussed in section 4.2. After that, the method of benchmarking will be described in section 4.3. Finally, the results of benchmarking will be shown and discussed in section 4.4.

4.1 Generating random partial Mealy machines

A set of partial Mealy machines was randomly generated in order to benchmark ComputeAllApartPairs. The algorithm that generated these partial Mealy machines, called GenerateMealyMachine, will be described in this section. GenerateMealyMachine is based on the algorithm presented in chapter 4 of 'On the Performance of Automata Minimization Algorithms' [1], which will be called GenerateNFA from now on.

GenerateNFA generates a nondeterministic finite automaton, whereas random partial Mealy machines are needed for the benchmarks. Therefore, there are some differences between GenerateNFA and GenerateMealy-Machine:

- GenerateMealyMachine assigns an output value to each generated transition, because a nondeterministic finite automaton does not have an output symbol associated to its transition, while a Mealy machine does.
- GenerateMealyMachine terminates when all states are visited once, whereas GenerateNFA continues to add random transitions. Generate-MealyMachine terminates at that point because it needs to generate a

partial Mealy machine that is a tree. Trees always have $|V| - 1$ edges, where V is the set of vertices.

The algorithm GenerateMealyMachine randomly generates one partial Mealy machine in rooted tree form, which is achieved by starting with an initial state and then adding random transitions. If a transition was not generated during the algorithm, then it is undefined in the resulting partial Mealy machine. GenerateMealyMachine expects as input variables three integers: one represents the size of the input alphabet, one the size of the output alphabet and one the number of states. The input and output alphabet are, respectively, the set of integers ranging from 0 to the size of the input and output alphabet minus one. Each state is represented by a unique natural number between 0 and the number of states minus one.

GenerateMealyMachine uses two arrays which are called 'visited' and 'unvisited'. The array 'visited' represents states that are already placed in the partial Mealy machine. The array 'unvisited' represents the states that still need to be added to the machine. At first, only the initial state is in the array 'visited' and the rest of the states are in 'unvisited'. The algorithm builds the machine by adding transitions with a random state from 'visited' as source and a random state from 'unvisited' as target. In this way, it is ensured that the generated partial Mealy machine is a tree. After a transition has been added, the target state is moved from 'unvisited' to 'visited', since it is part of the machine. The input character of this transition is chosen randomly from the set of input symbols for which the source state does not have an outgoing transition. This is to ensure that the resulting machine is deterministic. If the source node has an outgoing transition for each input character, then this source node is removed from 'visited' and a new source node is chosen instead. The output of a transition is chosen randomly from the output alphabet. The algorithm terminates when 'unvisited' is empty.

The symbols used in the pseudocode for this algorithm are similar to the symbols used to describe Mealy machines:

- I is an input alphabet.
- O is an output alphabet.
- Q is a set of states.
- $q_0 \in Q$ is the initial state of a Mealy machine.

Algorithm 5 GenerateMealyMachine(I, O, Q)

```
1: Let $\delta, \lambda$ be partial functions
2: Let $V, U$ be arrays
3: $V \leftarrow \{q_0\}$
4: $U \leftarrow Q \setminus \{q_0\}$
5: while $U \neq \emptyset$ do
6:     Choose $q_v \in V$ randomly
7:     while $\delta(q_v, i)\downarrow$ for each $i \in I$ do
8:         Remove $q_v$ from $V$
9:         Choose $q_v \in V$ randomly
10:    end while
11:    Choose $q_u \in U$ randomly
12:    Choose $i \in I$ randomly, such that $\delta(q_v, i)\uparrow$
13:    Choose $o \in O$ randomly
14:    $\delta(q_v, i) \leftarrow q_u$
15:    $\lambda(q_v, i) \leftarrow o$
16:    $U \leftarrow U \setminus \{q_u\}$
17:    $V \leftarrow V \cup \{q_u\}$
18: end while
19: return $(I, O, Q, q_0, \delta, \lambda)$
```

4.2 Influence of machine properties

The influence of the number of states and the input and output alphabet sizes of partial Mealy machines on the performance of ComputeAllApart-Pairs will be discussed in this section.

The complexity of ComputeAllApartPairs depends on both the number of states and the size of the input alphabet, as described in section 3.4.2. The time complexity of ComputeAllApartPairs scales quadratically in the number of states and linearly in the size of the input alphabet. However, the probability that both $\delta(p, i)$ and $\delta(q, i)$ are defined is smaller for machines that were generated with a larger input alphabet. This is because each transition gets assigned an input symbol that is chosen from a larger set of symbols. As a result, both for-loops of DetermineApart will be executed less often when the input alphabet is large, leading to better performance.

The size of the output alphabet that was used to generate a partial Mealy machine has an effect on the performance of the algorithm as well. If a partial Mealy machine was generated with a larger output alphabet, then there is a larger set of output symbols that can get assigned to each transition. As a result, the probability that two different transitions give the same output symbol decreases. This increases the likeliness that the for-loop at lines 7-12 of DetermineApart goes to line 10. Consequently, the algorithm makes fewer recursive calls, which should yield better performance.

4.3 Benchmark description

In this section, it will be described how the benchmarks were performed.

GenerateMealyMachine was used to generate partial Mealy machines for benchmarking. The number of states and alphabet sizes that were given as input to GenerateMealyMachine differ. The number of states varied between 2 and 50 and the alphabet sizes varied between 2 and 40. Twenty partial Mealy machines were generated per combination of number of states, input alphabet size and output alphabet size.

The performance of ComputeAllApartPairs was measured by recording the time it needed to terminate. Each combination of options of the variations described in section 3.5 was benchmarked. One variation decides in what order the states are picked in ComputeAllApartPairs and its options are: random, top-down and bottom-up. The other variation changes the order of the for-loops of DetermineApart and its two options are the standard order of the for-loops and the reversed order of the for-loops. For every combination of options of the two variations, ComputeAllApartPairs was performed a hundred times on each generated partial Mealy machine. The average of those hundred runs was calculated and will be referred to as the performance of that machine using those options of the variations. For every combination of number of states, input and output alphabet size and variation options, the average performance of the corresponding twenty machines was calculated. These averages are used in the graphs in this chapter and can be found in the tables in appendix A in the column ‘Average’. The performances of the best and worst performing machines of such a combination can be found, respectively, in the column ‘Minimum’ and the column ‘Maximum’.

The benchmarks were performed with an AMD Ryzen 5 2600 processor, NVIDIA GeForce RTX 2060 graphics card and 16 GB of RAM. The operating system that was used is Pop!_OS 22.04 LTS.

4.4 Benchmark results

The results of the benchmarks will be shown and discussed in this section. Tables in which the results of the benchmarks are presented can be found in appendix A.

The legends in figs. 4.1 to 4.3, 4.5 and 4.6 refer to the variations discussed in section 3.5. The terms ‘top-down’, ‘bottom-up’ and ‘random’ represent the options of the variation that determines in what order the pairs of states are chosen in ComputeAllApartPairs. The term ‘reversed’ indicates that DetermineApartReversed was used instead of DetermineApart. The performance of the fastest machine is used as the negative error and that of the slowest as the positive error in figs. 4.1 to 4.3, 4.5 and 4.6. The error bars in fig. 4.4 represent the performances of the best and worst performing machines averaged over all options of variations. In figs. 4.1 to 4.4 both the input and output alphabet have the same size.

Figures 4.1 to 4.4 show that ComputeAllApartPairs scales quadratically in the number of states in the given machine. In section 3.4.2, the time complexity of ComputeAllApartPairs was found to be $\mathcal{O}(|I|n^2)$. This means that the algorithm performs in accordance with its time complexity in terms of number of states.

Moreover, figs. 4.1 to 4.6 show that bottom-up performs better than random, which in turn performs better than top-down. It was predicted in section 3.5.1 that bottom-up would be the fastest option. The non-reversed variations perform better than the reversed variations. In section 3.5.2 it was stated that the reversed variations become more efficient as there are more pairs of apart states in the map A. As the map A is empty when ComputeAllApartPairs starts, it is to be expected that the reversed variations perform worse than the non-reversed variations.

Figures 4.4 and 4.5 show that ComputeAllApartPairs performs better on machines that were generated with larger input alphabet sizes, as was predicted in section 4.2. The differences between the variation options decrease as the alphabet input size increases, which can be seen in figs. 4.1 to 4.3 and 4.5. This is due to the fact that the for-loops of DetermineApart are executed less often when the input alphabet is larger. As was discussed in section 3.5, the difference in performance between variations is caused by the number of recursive calls that do not terminate at line 5 of DetermineApart. If the for-loop at lines 13-18 has a low probability of executing, then each variation will make fewer recursive calls, which ultimately leads to fewer recursive calls not terminating at line 5. This causes the difference between the variations to shrink.

Figure 4.6 shows that the performance of the non-reversed variations improves as the output alphabet size used to generate the partial Mealy machines increases, as was predicted in section 4.2. Figure 4.6 also shows that the reversed variations have equal performance for all output alphabet sizes. This is because the for-loop at lines 7-12 of DetermineApartReversed will be executed before the for-loop that checks whether there is an input symbol for which the states give different output symbols. As a result, DetermineApartReversed makes the same number of recursive calls regardless of the output alphabet size, which leads to nearly equal performance for all output alphabet sizes.

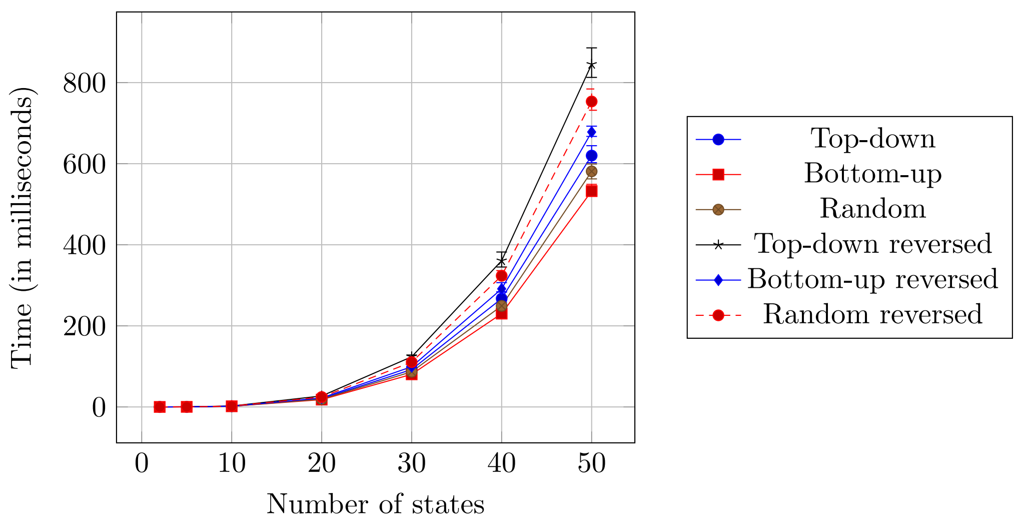

Figure 4.1: Performance of ComputeAllApartPairs on machines with alphabet size 2

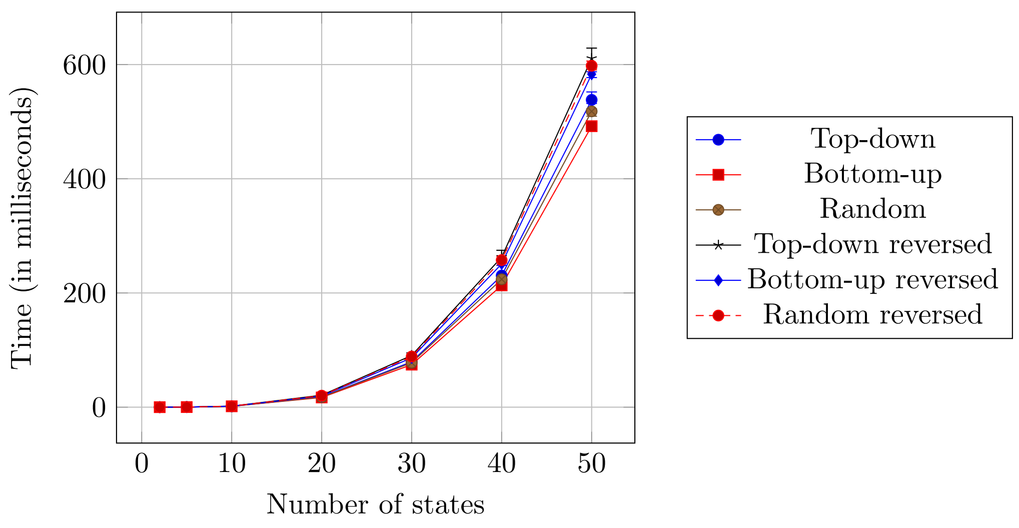

Figure 4.2: Performance of ComputeAllApartPairs on machines with alphabet size 5

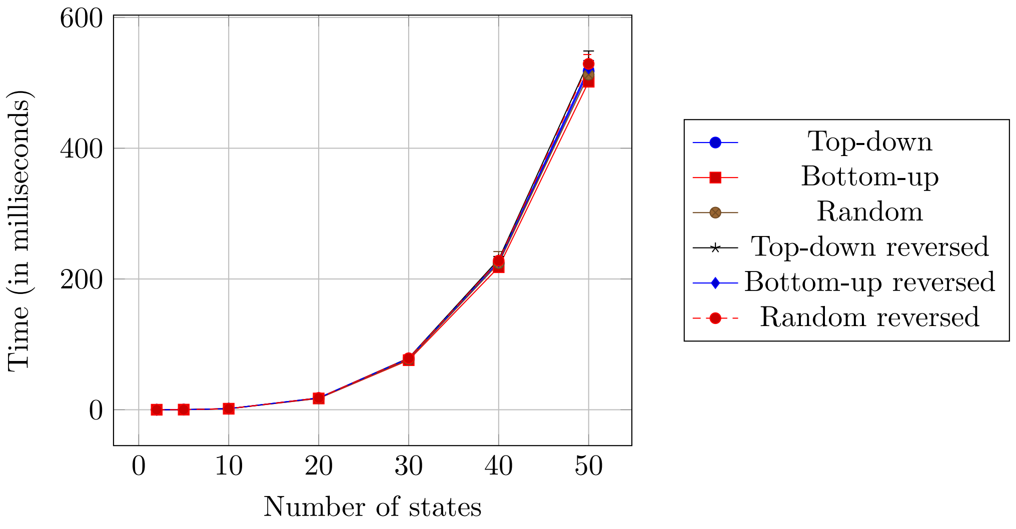

Figure 4.3: Performance of ComputeAllApartPairs on machines with alphabet size 20

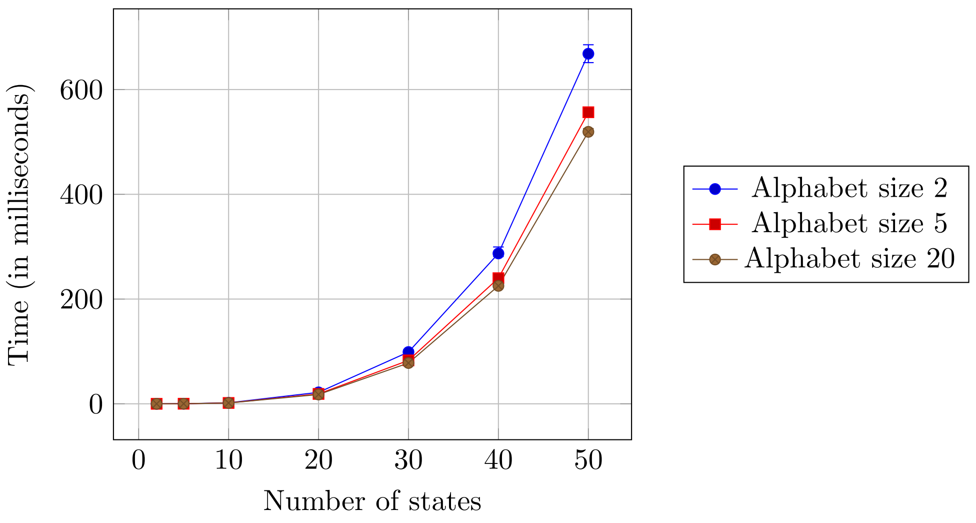

Figure 4.4: Average performance of variations of ComputeAllApartPairs

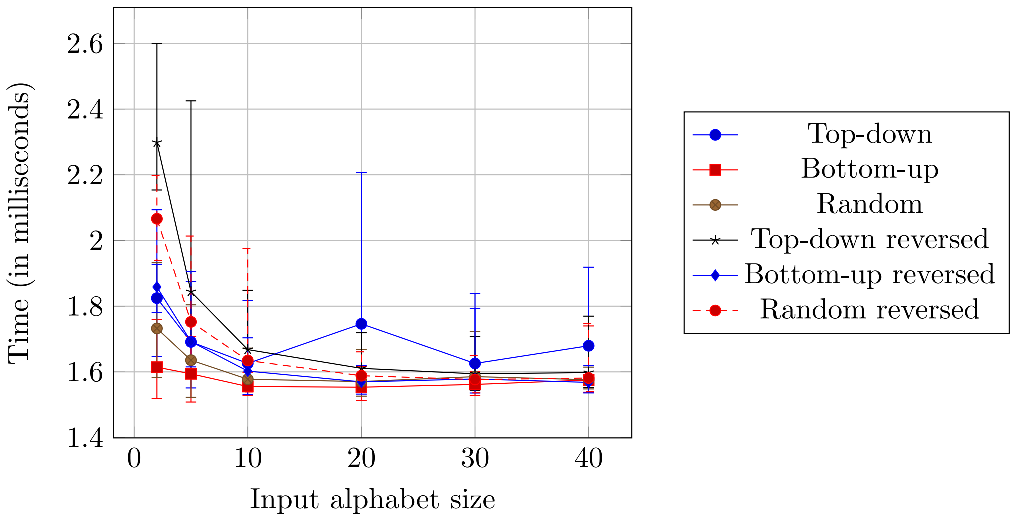

Figure 4.5: Varying input alphabet size (10 states, output alphabet size 2)

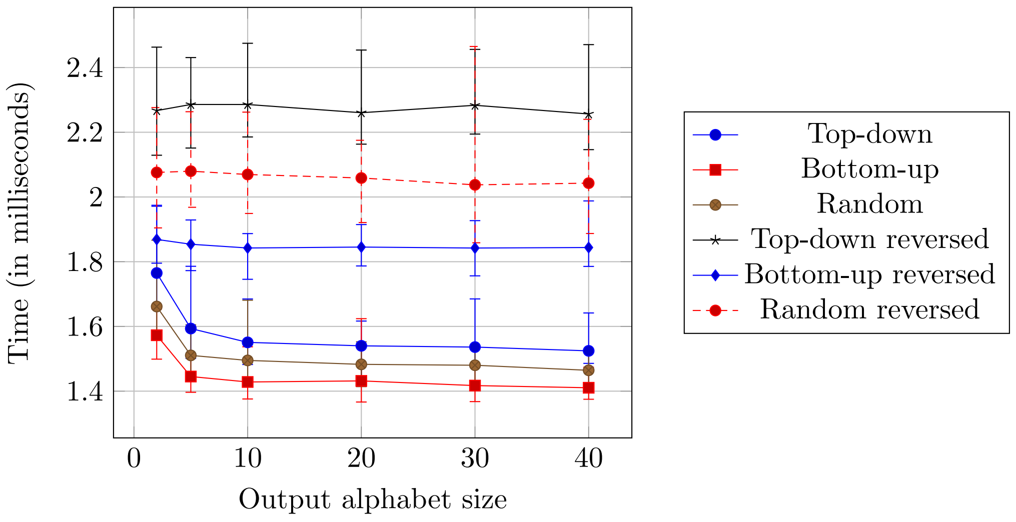

Figure 4.6: Varying output alphabet size (10 states, input alphabet size 2)

Chapter 5

Related Work

Research that has already been done on the subject of finding all apart states in Mealy machines will be discussed in this section.

An algorithm that computes the shortest distinguishing sequence for each pair of apart states in a Mealy machine has been presented in 'Minimal Separating Sequences for All Pairs of States' [6]. This algorithm makes use of the fact that the negation of apartness, which will be called language equivalence from now on, is an equivalence relation. Thus, it is possible to construct a partition that consists of sets of language equivalent states. The algorithm partitions the states of a Mealy machine into sets of language equivalent states, similar to the automata minimisation algorithms of Moore [5] and Hopcroft [3]. A splitting tree [4] is maintained to keep track of, for each split of a set of states, the sequence of input symbols that was used to split that set. After the algorithm has terminated, the distinguishing sequence of a pair of apart states can be found in this splitting tree. By splitting the sets in a certain order, the distinguishing sequences are guaranteed to be as short as possible. The algorithm has a time complexity of $\mathcal{O}(m \log n)$, where m is the number of transitions and n the number of states in a given Mealy machine.

While this is an efficient algorithm for finding all pairs of apart states and their corresponding distinguishing sequences in Mealy machines, it does not work for partial Mealy machines. This is due to the fact that no partition can be created that contains sets of language equivalent states, as language equivalence is not transitive in partial Mealy machines. The lack of transitivity is caused by the partiality of partial Mealy machines, as shown in example 5.1. Since the algorithms of Moore and Hopcroft are also based on creating a partition of sets of language equivalent states, they cannot be used either to find all pairs of apart states in a partial Mealy machine. Therefore, a different algorithm was needed that is applicable to partial Mealy machines.

Example 5.1. *In fig. 5.1 we have that q_0 and q_2 are not apart, q_2 and q_1*

are not apart, but q_0 and q_1 are apart.

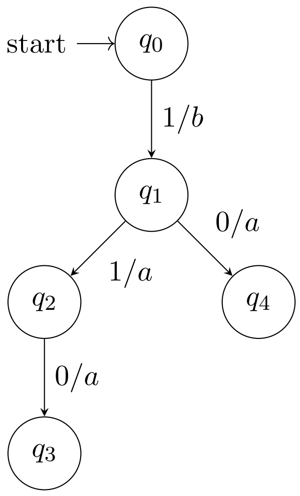

Figure 5.1: A slight variation of the partial Mealy machine in fig. 2.3 ($I = \{0,1\}, O = \{a,b\}$)

An algorithm that finds all pairs of apart states in a partial Mealy machine was implemented for the benchmarks in the paper 'A New Approach for Active Automata Learning Based on Apartness' [8]. This is because the $L^{\#}$ algorithm needs all pairs of apart states in a partial Mealy machine, as described in chapter 1. However, the implemented algorithm that finds all pairs of apart states was not discussed separately, which is why this thesis presents, analyses and benchmarks such an algorithm.

Chapter 6

Conclusions

An algorithm that finds all pairs of apart states in a partial Mealy machine was presented in this thesis. Its time complexity was found to be $\mathcal{O}(|I|n^2)$, where I is the input alphabet and n is the number of states of a partial Mealy machine. Two variations of the default algorithm were discussed. One determines the order in which ComputeAllApartPairs picks pairs of states and the other decides in what order the for-loops of DetermineApart are executed.

Benchmarks were performed on the algorithm which indicate that it scales quadratically in the number of states, which is in line with its time complexity. The benchmarks also showed that, when the number of states is fixed, the algorithm runs faster on partial Mealy machines in which the transitions use more input symbols. ComputeAllApartPairs performs better when the states are traversed bottom-up instead of top-down or randomly and swapping the for-loops of DetermineApart worsened the performance of ComputeAllApartPairs.

Possible future work on this topic would include transforming the presented algorithm into algorithms that are suitable for other types of finite-state machines with partial transition and output functions. The performance of these transformed algorithms could then be compared to that of the original algorithm. Also, the current implementation could be improved, as there may be data structures and low-level optimisations that improve the algorithm's performance.

Bibliography

[1] Marco Almeida, Nelma Moreira, and Rogério Reis. On the performance of automata minimization algorithms. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, pages 3–14. Springer, 2008. Contributed talk.

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

[3] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[4] D. Lee and M. Yannakakis. Testing finite-state machines: state identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 1994.

[5] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies. (AM-34)*, 34:129–153, 1956.

[6] Rick Smetsers, Joshua Moerman, and David N. Jansen. Minimal separating sequences for all pairs of states. In Adrian-Horia Dediu, Jan Janoušek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, pages 181–193, Cham, 2016. Springer International Publishing.

[7] Frits Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, jan 2017.

[8] Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. *CoRR*, abs/2107.05419, 2021.

Appendix A

Benchmarking result tables

The column 'Number of states' refers to the number of states in a benchmarked partial Mealy machine and the columns 'Input alphabet' and 'Output alphabet' refer to the sizes of its input and output alphabets. Per combination of number of states, size of input alphabet and size of output alphabet, twenty partial Mealy machines were benchmarked. For each combination of options of the variations, ComputeAllApartPairs was called a hundred times on each of the partial Mealy machines. The average time ComputeAllApartPairs needed to terminate over the hundred runs on a partial Mealy machine will be called the performance of that partial Mealy machine for that combination of options of variations. Each table contains the performance of the partial Mealy machines using one specific combination of variation options. The column 'Average' represents the average performance in milliseconds of the twenty partial Mealy machines corresponding to that combination of number of states, size of input alphabet and size of output alphabet. The columns 'Minimum' and 'Maximum' represent the performances in milliseconds of the partial Mealy machines that were, respectively, the fastest and the slowest over their hundred runs out of the twenty corresponding partial Mealy machines.

The terms 'top-down', 'bottom-up' and 'random' refer to the order in which the states are chosen in ComputeAllApartPairs. DetermineApart was used if the term 'reversed' is not in the name of a variation and DetermineApartReversed was used if the term 'reversed' is in the name of a variation.

Number of states	Input alphabet	Output alphabet	Average	Minimum	Maximum
2	2	2	0.024468	0.01224	0.03248
5	2	2	0.220038	0.17242	0.30237
10	2	2	1.88057	1.66368	2.12626
20	2	2	20.2872	18.9555	21.6695
30	2	2	92.4129	87.176	99.6594
40	2	2	267.336	257.738	283.081
50	2	2	619.967	602.466	644.381
2	5	5	0.008881	0.0081	0.01014
5	5	5	0.168481	0.15603	0.22902
10	5	5	1.56638	1.5176	1.6659
20	5	5	17.9844	17.6069	18.9005
30	5	5	79.5247	78.3154	80.8282
40	5	5	230.511	228.541	232.357
50	5	5	538.15	531.835	552.052
2	20	20	0.008263	0.00801	0.00996
5	20	20	0.187612	0.16311	0.23925
10	20	20	1.58529	1.55919	1.76057
20	20	20	17.5751	17.3846	17.7952
30	20	20	77.679	77.0436	78.8663
40	20	20	224.199	222.628	228.071
50	20	20	518.012	514.372	521.915
10	2	2	1.76834	1.6273	1.99874
10	5	5	1.56838	1.52082	1.64371
10	10	10	1.55642	1.52738	1.61369
10	20	20	1.54906	1.51305	1.58247
10	30	30	1.56207	1.53883	1.73744
10	40	40	1.55441	1.54028	1.56853
10	2	2	1.82455	1.64633	2.09347
10	5	2	1.69121	1.55108	1.87466
10	10	2	1.62573	1.56038	1.81747
10	20	2	1.7462	1.55769	2.20642
10	30	2	1.62556	1.55947	1.83886
10	40	2	1.67947	1.56054	1.91867
10	2	2	1.76503	1.66199	1.97439
10	2	5	1.59349	1.50099	1.7858
10	2	10	1.55073	1.48227	1.6848
10	2	20	1.54005	1.44502	1.61662
10	2	30	1.53618	1.47327	1.68503
10	2	40	1.52453	1.4858	1.64163

Table A.1: Benchmark results of ‘top-down’ variation

Number of states	Input alphabet	Output alphabet	Average	Minimum	Maximum
2	2	2	0.0086275	0.00801	0.01293
5	2	2	0.191601	0.162	0.2643
10	2	2	1.60685	1.52135	1.71012
20	2	2	18.0528	17.2952	19.1853
30	2	2	80.4229	77.7583	85.156
40	2	2	230.216	224.433	243.368
50	2	2	532.388	519.714	549.029
2	5	5	0.0095825	0.00801	0.01192
5	5	5	0.160987	0.15356	0.17371
10	5	5	1.51725	1.47208	1.5999
20	5	5	16.9904	16.4859	17.5179
30	5	5	74.4891	73.4082	75.5436
40	5	5	213.267	210.464	216.047
50	5	5	492.073	484.003	498.345
2	20	20	0.0080635	0.00801	0.00829
5	20	20	0.162988	0.15705	0.17985
10	20	20	1.53792	1.51233	1.56287
20	20	20	17.2467	17.0382	17.6278
30	20	20	75.8291	75.152	76.3859
40	20	20	218.033	216.835	220.087
50	20	20	501.758	497.505	504.648
10	2	2	1.58557	1.48733	1.6937
10	5	5	1.50824	1.44716	1.60837
10	10	10	1.53572	1.49805	1.60803
10	20	20	1.54954	1.4837	1.68476
10	30	30	1.54651	1.49618	1.57182
10	40	40	1.55657	1.52559	1.62614
10	2	2	1.61431	1.51839	1.75968
10	5	2	1.59407	1.50815	1.70717
10	10	2	1.55538	1.52829	1.599
10	20	2	1.55332	1.51298	1.6168
10	30	2	1.56175	1.52762	1.60081
10	40	2	1.57528	1.53805	1.73984
10	2	2	1.57279	1.49887	1.65385
10	2	5	1.44514	1.39644	1.51317
10	2	10	1.42836	1.37582	1.53711
10	2	20	1.43168	1.36644	1.62396
10	2	30	1.41721	1.36772	1.47591
10	2	40	1.41049	1.37494	1.47406

Table A.2: Benchmark results of ‘bottom-up’ variation

Number of states	Input alphabet	Output alphabet	Average	Minimum	Maximum
2	2	2	0.0088755	0.00809	0.01007
5	2	2	0.178397	0.15875	0.21151
10	2	2	1.66144	1.55426	1.8064
20	2	2	19.2393	17.7695	20.1182
30	2	2	86.784	82.3473	92.3702
40	2	2	249.256	242.163	261.436
50	2	2	581.086	562.52	599.208
2	5	5	0.008029	0.00801	0.00814
5	5	5	0.167481	0.15737	0.24096
10	5	5	1.54685	1.50059	1.64831
20	5	5	17.6535	17.0672	18.9392
30	5	5	77.8321	77.0011	78.7296
40	5	5	223.987	220.599	226.455
50	5	5	518.043	509.549	522.73
2	20	20	0.0080315	0.00801	0.00811
5	20	20	0.171455	0.15763	0.25535
10	20	20	1.55614	1.52787	1.58136
20	20	20	17.5006	17.3127	17.8064
30	20	20	77.1979	76.7825	77.7613
40	20	20	224.137	221.131	241.982
50	20	20	512.56	507.569	518.869
10	2	2	1.67882	1.56806	1.84364
10	5	5	1.55523	1.49559	1.70315
10	10	10	1.55669	1.51617	1.64158
10	20	20	1.56163	1.49943	1.71949
10	30	30	1.55568	1.53548	1.57163
10	40	40	1.56603	1.54166	1.63381
10	2	2	1.7323	1.58331	1.93214
10	5	2	1.63558	1.52272	1.80412
10	10	2	1.57756	1.54777	1.67233
10	20	2	1.57052	1.52607	1.66814
10	30	2	1.58553	1.55348	1.72237
10	40	2	1.57611	1.54804	1.61445
10	2	2	1.66155	1.58215	1.86641
10	2	5	1.51051	1.45538	1.60506
10	2	10	1.49499	1.43677	1.68088
10	2	20	1.48281	1.41394	1.55252
10	2	30	1.48006	1.41476	1.54002
10	2	40	1.46442	1.40762	1.51973

Table A.3: Benchmark results of ‘random’ variation

Number of states	Input alphabet	Output alphabet	Average	Minimum	Maximum
2	2	2	0.009735	0.00819	0.01254
5	2	2	0.211189	0.19459	0.23288
10	2	2	2.29643	2.16138	2.44967
20	2	2	27.0647	26.0993	28.8154
30	2	2	123.728	115.087	128.058
40	2	2	360.846	345.166	382.157
50	2	2	845.314	812.856	885.71
2	5	5	0.012026	0.00801	0.01325
5	5	5	0.187421	0.15685	0.32366
10	5	5	1.80857	1.68207	2.0518
20	5	5	20.9106	20.1632	21.9978
30	5	5	90.5254	87.0448	95.0894
40	5	5	262.727	253.405	274.751
50	5	5	610.777	593.25	628.857
2	20	20	0.0080645	0.00801	0.00832
5	20	20	0.163541	0.15613	0.18214
10	20	20	1.62482	1.5408	1.72064
20	20	20	18.1084	17.5293	18.6757
30	20	20	79.1871	77.7756	80.6432
40	20	20	229.867	226.07	234.434
50	20	20	531.946	525.962	548.723
10	2	2	2.30901	2.19466	2.52216
10	5	5	1.80847	1.67957	1.99979
10	10	10	1.66404	1.5728	1.78052
10	20	20	1.62675	1.53203	1.82356
10	30	30	1.58262	1.53963	1.70441
10	40	40	1.59057	1.53393	1.82469
10	2	2	2.29845	2.15372	2.60029
10	5	2	1.84413	1.68583	2.42512
10	10	2	1.6678	1.54549	1.84848
10	20	2	1.6107	1.54134	1.71912
10	30	2	1.59413	1.54388	1.70788
10	40	2	1.59789	1.55159	1.76959
10	2	2	2.26689	2.12899	2.46305
10	2	5	2.28584	2.15125	2.43084
10	2	10	2.28577	2.18549	2.47493
10	2	20	2.26035	2.16322	2.45406
10	2	30	2.28338	2.19439	2.45606
10	2	40	2.25647	2.14631	2.47082

Table A.4: Benchmark results of ‘top-down reversed’ variation

Number of states	Input alphabet	Output alphabet	Average	Minimum	Maximum
2	2	2	0.009974	0.00802	0.01297
5	2	2	0.184112	0.17547	0.2066
10	2	2	1.85672	1.81443	1.91611
20	2	2	22.1786	21.7342	22.7086
30	2	2	99.2359	97.4059	101.03
40	2	2	291.284	284.187	306.607
50	2	2	678.09	667.257	692.562
2	5	5	0.0080425	0.00801	0.00823
5	5	5	0.169002	0.15549	0.20186
10	5	5	1.67825	1.5947	1.85901
20	5	5	19.5028	18.9582	20.07
30	5	5	86.351	85.0713	87.8666
40	5	5	250.93	248.2	253.768
50	5	5	583.131	577.233	587.351
2	20	20	0.0081615	0.00801	0.00956
5	20	20	0.195778	0.17384	0.24873
10	20	20	1.65729	1.58727	1.85046
20	20	20	18.1373	17.7657	18.5821
30	20	20	78.6814	77.3726	81.3708
40	20	20	226.849	223.913	229.842
50	20	20	522.333	515.913	534.089
10	2	2	1.87162	1.81004	2.07893
10	5	5	1.6838	1.61562	1.77832
10	10	10	1.60103	1.54472	1.6633
10	20	20	1.59722	1.53087	1.85452
10	30	30	1.56504	1.53502	1.63186
10	40	40	1.56532	1.5365	1.6037
10	2	2	1.85841	1.78116	1.92656
10	5	2	1.69219	1.61517	1.90508
10	10	2	1.60237	1.53168	1.70366
10	20	2	1.56954	1.53227	1.61887
10	30	2	1.57843	1.53566	1.79331
10	40	2	1.56794	1.53594	1.61913
10	2	2	1.86891	1.79539	1.97136
10	2	5	1.85396	1.77249	1.92897
10	2	10	1.84238	1.74562	1.88686
10	2	20	1.84522	1.78689	1.91475
10	2	30	1.84225	1.75633	1.9269
10	2	40	1.84387	1.78543	1.98782

Table A.5: Benchmark results of ‘bottom-up reversed’ variation

Number of states	Input alphabet	Output alphabet	Average	Minimum	Maximum
2	2	2	0.009874	0.00813	0.01291
5	2	2	0.198064	0.17494	0.23939
10	2	2	2.05997	1.9365	2.19807
20	2	2	24.4418	23.3911	25.355
30	2	2	110.578	106.269	114.705
40	2	2	323.541	313.938	336.219
50	2	2	753.468	731.832	784.302
2	5	5	0.0082295	0.00811	0.00896
5	5	5	0.179931	0.15657	0.24573
10	5	5	1.73158	1.61251	1.93206
20	5	5	20.1576	19.6401	20.8862
30	5	5	88.7017	86.3991	91.3077
40	5	5	257.571	252.142	264.679
50	5	5	597.868	590.711	606.139
2	20	20	0.008476	0.00801	0.00994
5	20	20	0.163943	0.1578	0.18378
10	20	20	1.60009	1.53346	1.66726
20	20	20	18.0624	17.5773	18.4548
30	20	20	78.7158	77.2828	80.7656
40	20	20	228.03	223.918	231.493
50	20	20	529.048	522.856	543.326
10	2	2	2.07664	1.93472	2.26079
10	5	5	1.74754	1.6317	1.88415
10	10	10	1.63167	1.55962	1.71743
10	20	20	1.6149	1.53626	1.80393
10	30	30	1.56819	1.54001	1.66584
10	40	40	1.59029	1.53497	1.70455
10	2	2	2.06614	1.93962	2.19732
10	5	2	1.75221	1.64245	2.0135
10	10	2	1.63465	1.539	1.97558
10	20	2	1.58828	1.538	1.66099
10	30	2	1.57676	1.53621	1.64954
10	40	2	1.58118	1.54062	1.74656
10	2	2	2.07585	1.90446	2.27624
10	2	5	2.07974	1.96811	2.26348
10	2	10	2.06951	1.94902	2.26218
10	2	20	2.05864	1.92097	2.17518
10	2	30	2.03751	1.85853	2.46505
10	2	40	2.04273	1.88713	2.23996

Table A.6: Benchmark results of ‘random reversed’ variation