Minimization of Symmetric Difference Finite Automata

Graham Müller

Thesis presented in partial fulfilment
of the requirements for the degree of
Master of Science
At the University of Stellenbosch

Dr. L. van Zijl
December 2005

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to be the NRF.

Contents

Chapter 1

Introduction

Traditional finite automata (FAs) have existed in the literature since the early 1930s [15]. These FAs represent basic machines which consist of states and rules which govern which states are active at any point in time. The outputs of these machines are to either accept or reject a given input sequence. FAs are divided into two classes, namely deterministic FAs (DFAs) and non-deterministic FAs (NFAs).

As FAs are easy to understand and manipulate they have received a great deal of attention. Many applications, particularly in the field of natural language processing where pattern recognition [4] is of importance, feature large FAs [4]. These large FAs are constructed for simplicity, not efficiency, and as a result FAs may exist which return the exact same results but consist of fewer states. These smaller FAs can be constructed from the original FA by removing or combining states within the original FA. Minimization algorithms for both DFAs and NFAs have been constructed to detect and remove these removable states [19] and produce the smallest possible FA which returns the same results. These minimization algorithms play an essential role in many large scale practical applications to improve the efficiency of the applications.

Recently a new form of FAs has been derived from the standard FA form, namely the generalized FAs ($*$-FAs)[17]. These $*$-FAs generalize the traditional FA by allowing any binary associative commutative operator to be used to control the transitions of states instead of the union rule used by traditional NFAs. Traditional NFAs are in fact a specialized form of $*$-FAs.

We are interested in one form of the $*$-FAs specifically, namely the symmetric difference FAs ($\oplus$-FAs) which make use of the symmetric difference operator ($\oplus$). Practical applications have been developed for $\oplus$-FAs such as random number generation [18] and hashing [14]. Furthermore, these $\oplus$-FA have been shown to be at least as descriptive as traditional NFAs with the same number of states [18] implying a potential for further such practical applications for $\oplus$-FAs.

As with traditional FAs it would be beneficial if applications using a large number of states could be minimized in terms of its number of states. However, as these $\oplus$-FAs are still relatively young, little research has been done into the minimization of $\oplus$-FAs. In this regard we are faced with a blank slate to construct $\oplus$-FA minimization algorithms. We plan to investigate traditional FA minimization algorithms and properties of $\oplus$-FAs to base $\oplus$-FA minimization algorithms on.

We start this thesis by providing basic definitions of the traditional FAs in chapter 2. We also define properties of these FAs such as equivalence and minimality of FAs, and discuss the relationship between DFAs and NFAs. This is done both to recap on the definitions of traditional FAs, but also to define the notation we will be using throughout the thesis.

In chapter 3 we provide the definitions and properties of FAs for $*$-FAs based on the definitions and properties from chapter 2. Before discussing and presenting the formal definitions of $\oplus$-FAs we discuss an example of $*$-FAs which use the intersection operator ($\cap$-FA).

We move onto traditional FA minimization algorithms in chapter 4. This chapter is used to provide the basic format of FA minimization algorithms and as a basis for $\oplus$-FA minimization algorithms. We divide the discussion of minimization algorithms for traditional FAs into three sections. In the first section we discuss reducible states (these are states which can be removed from the FA without affecting the input sequences accepted by the FA.). We also define three distinct categories of FA minimization algorithms.

The second section of chapter 2 deals with DFA minimization. The minimization algorithms developed for DFAs are simpler and faster than those for NFAs due to the simpler nature of DFAs. We discuss three DFA minimization algorithms which correspond to the three categories of FA minimization algorithms. These minimization algorithms are by Daciuk, Watson and Watson [4]; Hopcroft [7] and Brzozowski [2].

In the third section of chapter 2 we discuss an NFA minimization algorithm and an NFA reduction algorithm. A reduction algorithm does not always result in the smallest possible FA but runs faster than minimization algorithms. NFA reduction algorithms are of importance as it is shown in [10] that the problem of NFA minimization is NP hard. The NFA minimization algorithm we discuss is by Kameda and Weiner [11] and the reduction algorithm by Ilie and Yu[9]. Similar to how we wish to base $\oplus$-FA minimization algorithms on the NFA minimization algorithms, the NFA minimization and reduction algorithms are based on the DFA minimization algorithms by Brzozowski and Hopcroft respectively.

With the DFA and NFA minimization algorithms from chapter 4 in mind we approach the problem of $\oplus$-FA minimization in chapter 5. After discussing properties of $\oplus$-FAs in terms of minimization we develop a

minimization algorithm for unary[1] $\oplus$-FAs based on their relationship with linear feedback shift registers (LFSRs) [14]. We show that the algorithm is efficient, however, it is not extendable to non unary $\oplus$-FAs. After this we move onto developing minimization and reduction algorithms for general $\oplus$-FAs. We approach the problem of $\oplus$-FA minimization from the three categories of minimization algorithms. The first algorithm we discuss minimizes $\oplus$-FAs and is based on the NFA minimization algorithm by Kameda and Weiner. Next we construct a transformation called the subset $\oplus$ transformation which provides a method to reduce both the number of states and the number of transitions of a $\oplus$-FA and provides certain properties of $\oplus$-FAs such as generating equivalent $\oplus$-FAs with the same number of states.

In chapter 6, we move from the minimization problem of the previous chapters to discuss the problem of storing $\oplus$-FAs. Although efficient storage algorithms exist for traditional FAs they take advantage of the union rule and so are not necessarily directly to applicable to $\oplus$-FAs. These storage formats need to be able to be both stored efficiently and be run without excessive computational overhead. We propose two formats for $\oplus$-FAs. The first, for unary $\oplus$-FAs, is based on the unary $\oplus$-FA minimization algorithm from chapter 5. This algorithm provides both a good compression ratio and no computational overhead when executing the $\oplus$-FA. The second format we discuss is designed for use on general $\oplus$-FAs. It is adapted for use on $\oplus$-FAs from the DFA storage format used by Daciuk [4]. We begin the discussion of this format with the basic format and its execution after which we provide compression algorithms which can be applied to the basic format.

We provide an appendix which contains terminology, notations and definitions which we use throughout this thesis.

[1] Unary $\oplus$-FAs has an alphabet of size 1.

Chapter 2

Preliminaries: Finite Automata

In this chapter we discuss background knowledge of decision theory. We provide this chapter not only to refresh concepts of decision theory, but also to provide the notations that we shall be using throughout this thesis.

We start by defining finite automata and then discuss their properties, characteristics and general transformations on these automata. The notations and definitions discussed are derived from Sipser [15] with some concepts taken from Kameda and Weiner [11].

In section 2.1 we will show the various general representations of automata used in this thesis. Thereafter, we discuss some extensions on these definitions used to define equivalence of finite automata. We finish this chapter by discussing the subset transformation which transforms NFAs into an equivalent DFA.

2.1 Representations

Definition 2.1

*A **finite automaton** (FA) is defined as a 5-tuple $M = (Q, \Sigma, \delta, Q_0, F)$ where*

- *Q is the finite, non-empty set of **states**,*
- *Σ is the finite, non-empty set called the **alphabet**,*
- *$\delta : Q \times \Sigma \to 2^Q$ is the **transition function**,*
- *$Q_0 \subseteq Q$ is the set of **initial states**, and*
- *$F \subseteq Q$ is the set of **final states**.*

A finite automaton is a machine which takes as input a string of symbols from its alphabet and either accepts or rejects this inputted string. The FA keeps track of a set of active states known as the *active state set*. The active state set initially contains only the initial states. The active state set is updated according to the next symbol in the input string and the transition function. A state which is currently in the active state set is defined as *active*. The input string is accepted if, after all the symbols in the input string have been processed, there is at least one final state in the active set. Likewise if there is no final state in the active state set, the input string is rejected.

Consider the following example.

Example 2.2 *Let* $M = (Q, \Sigma, \delta, Q_0, F)$ *be a FA with*

- $Q = \{q_0, q_1, q_2\}$,
- $\Sigma = \{0, 1\}$,
- δ *is described as:*

$$\begin{array}{lcllcl} \delta(q_0, 0) & = & \{q_0\} & , \ \delta(q_0, 1) & = & \{q_2\} \\ \delta(q_1, 0) & = & \{q_0, q_1\} & , \ \delta(q_1, 1) & = & \{q_2\} \\ \delta(q_2, 0) & = & \{q_1\} & , \ \delta(q_2, 1) & = & \{q_2\} \ , \end{array}$$

- $Q_0 = \{q_0\}$, *and*
- $F = \{q_2\}$.

The initial set of states for the FA M *is* $\{q_0\}$. *The active state set remains as* $\{q_0\}$ *on input character* 0, *while on input character* 1 *the active state set becomes* $\{q_2\}$. *As* $\{q_2\}$ *is a final state, any input strings of any number of* 0*'s followed by a* 1 *are accepted by the FA* M. *When the state set* $\{q_2\}$ *is active, on input character* 0 *the active state set becomes* $\{q_1\}$. *The active state set of* $\{q_1\}$ *becomes* $\{q_0, q_1\}$ *on input symbol* 0. *Further examination of the transition function shows us that any input of* 1 *results in the active state set* $\{q_2\}$ *while inputs of* 0 *result in a state set of* $\{q_0\}, \{q_1\}$ *or* $\{q_0, q_1\}$.

FAs in the above format are not always easy to understand or follow. We present a format for representing FAs in graphical format and for representing the transition function in table format. A FA can also be represented graphically as follows:

- states are represented by circles,
- transitions are represented by directed arrows (where $q_i \in \delta(q_j, \sigma)$ a directed edge is drawn from q_j to q_i with the symbol σ as edge label),

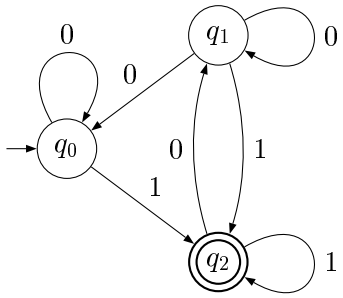

Figure 2.1: Finite Automaton Graph for Example 2.2.

- initial states are represented with an inbound edge with the base of the arrow not connected to any state, and
- final states are represented as double circles.

The graphical representation for the FA in example 2.2 above is given in figure 2.1.

Furthermore, we can represent the transition function as a transition table of states by alphabet symbols, where the table entries are subsets of the set of states. For example 2.2, the table would be:

	0	1
q_0	$\{q_0\}$	$\{q_2\}$
q_1	$\{q_0, q_1\}$	$\{q_2\}$
q_2	$\{q_1\}$	$\{q_2\}$

.

We often need to refer to the states reachable after one input symbol from another state. For this reason, we define children and parent states as follows:

Definition 2.3 *Given a FA* $M = (Q, \Sigma, \delta, Q_0, F)$*, the* ***children*** $ch(q)$ *of a state* $q \in Q$ *are*

$$ch(q) = \{q_i | q_i \in Q,\ \exists \sigma \in \Sigma,\ q_i \in \delta(q, \sigma)\},\ \textit{and}$$

the ***parents*** $pa(q)$ *of a state* $q \in Q$ *are*

$$pa(q) = \{q_i | q_i \in Q,\ \exists \sigma \in \Sigma,\ q \in \delta(q_i, \sigma)\}.$$

The parents and children of the states in the FA M in example 2.2 are:

$$\begin{array}{lcll} ch(q_0) & = & \{q_0, q_2\} & , \\ ch(q_1) & = & \{q_0, q_1, q_2\} & , \\ ch(q_2) & = & \{q_1, q_2\} & , \end{array}$$

$$\begin{array}{lcll} pa(q_0) & = & \{q_0, q_1\} & , \\ pa(q_1) & = & \{q_1, q_2\} & , \text{ and} \\ pa(q_2) & = & \{q_0, q_1, q_2\} & . \end{array}$$

This concludes the basic definitions of the FA we shall be using. In the next section we discuss the properties of FAs needed to determine whether two FAs accept the same input strings.

2.2 Equivalence

The FA resulting from a minimization algorithm is the smallest FA which accepts the same input strings as the original FA. In this section we discuss various properties of FAs which are used to compare FAs.

As we discussed in the previous section, a FA either accepts or rejects a string consisting of symbols from its alphabet. The set of all the word accepted by the FA is defined as the language of the FA as follows:

Definition 2.4

[Language] The ***language*** $\mathcal{L}(M)$ *accepted by a FA* M*, is the set of all strings accepted by the FA.*

For example, for the FA M in example 2.2 (page 7), $\mathcal{L}(M)$ is the set of all words consisting of 0's and 1's, ending in 1. Therefore $010 \notin \mathcal{L}(M)$ while $001 \in \mathcal{L}(M)$.

We can define the language of a FA based on the transition function. To do this we need the following extension of the transition function:

Definition 2.5 (Transition Function Extension)

The domain and range of the transition function δ *as defined in definition 2.1 can be extended as follows:*

- $\delta : 2^Q \times \Sigma \to 2^Q$ *is defined with* $P \subseteq Q$ *and* $\sigma \in \Sigma$ *as*

$$\delta(P, \sigma) = \bigcup_{q \in P} \delta(q, \sigma), \text{ and}$$

- $\delta : Q \times \Sigma^* \to 2^Q$ *is defined recursively with* $q \in Q$*,* $\sigma \in \Sigma$ *and* $x \in \Sigma^*$ *as*

$$\begin{array}{rcl} \delta(q, \epsilon) & = & \{q\} \\ \delta(q, \sigma x) & = & \bigcup_{q' \in \delta(q,\sigma)} \delta(q', x) \ . \end{array}$$

Using the above extensions we have that the language of an automaton $M = (Q, \Sigma, \delta, Q_0, F)$ is defined as

$$\mathcal{L}(M) = \{x | x \in \Sigma^*,\ \delta(Q_0, x) \cap F \neq \emptyset\}.$$

The language $\mathcal{L}(M)$ for the FA M in example 2.2 written as a regular expression [1] is

$$\mathcal{L}(M) = (0|1)^*1\ .$$

The extensions to the transition function allow us to extend the definition of children of states to include sets as follows:

$$ch(S) = \{q_i | q_i \in Q,\ \exists\sigma \in \Sigma,\ q_i \in \delta(S, \sigma)\},\ \text{where } S \subseteq Q\ .$$

Although a multitude of FAs exist, not all FAs have unique languages. It is this property that we take advantage of in FA minimization algorithms. We define FAs which accept the same language as equivalent.

Definition 2.6 (Equivalence)
The FAs M and M' are ***equivalent*** *iff*

$$\mathcal{L}(M) = \mathcal{L}(M').$$

We write $M \equiv M'$ if M is equivalent to M'.

This means that with $M = (Q, \Sigma, \delta, Q_0, F)$ and $M' = (Q', \Sigma, \delta', Q_0', F')$, M and M' are equivalent if for every word $x \in \Sigma^*$

$$\delta(Q_0, x) \cap F \neq \emptyset \Leftrightarrow \delta'(Q_0', x) \cap F' \neq \emptyset.$$

and

$$\delta(Q_0, x) \cap F = \emptyset \Leftrightarrow \delta'(Q_0', x) \cap F' = \emptyset\ .$$

Now that we have the equivalence relation for FAs, we can define a minimal FA as the FA with the smallest number of states which accepts a given language. We discuss the various aspects of minimality in more detail in chapter 3. Formally we define a FA as minimal as follows:

Definition 2.7 (Minimality)
A FA $M = (Q, \Sigma, \delta, Q_0, F)$ is ***minimal*** *if and only if for every FA $M' = (Q', \Sigma, \delta', Q_0', F')$ where $\mathcal{L}(M) = \mathcal{L}(M')$, it holds that*

$$|Q| \leq |Q'|.$$

Note 1 *Definition 2.7 contains an inequality implying that a minimal FA is not necessarily unique.*

[1] See convention A.8, page 170.

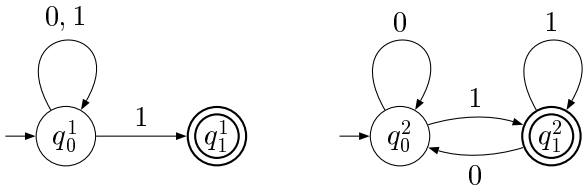

Figure 2.2: Example 2.8. Left: M_1. Right: M_2

Consider the following example of minimal FAs and uniqueness.

Example 2.8 (Uniqueness)

Take the language from example 2.2, that is $\mathcal{L}(M) = (0|1)^*1$.

Let $M_1 = (\{q_0^1, q_1^1\}, \{0,1\}, \delta_1, \{q_0^1\}, \{q_1^1\})$ *with* δ_1 *given by (also shown in figure 2.2)*

	0	1
q_0^1	$\{q_0^1\}$	$\{q_0^1, q_1^1\}$
q_1^1	$\emptyset$	$\emptyset$

.

Let $M_2 = (\{q_0^2, q_1^2\}, \{0,1\}, \delta_2, \{q_0^2\}, \{q_1^2\})$ *with* δ_2 *given by (also shown in figure 2.2)*

	0	1
q_0^2	$\{q_0^2\}$	$\{q_1^2\}$
q_1^2	$\{q_0^2\}$	$\{q_1^2\}$

.

In the FAs above we have that the states q_1^1 *and* q_1^2 *are final states and active only on input* 1*. The FAs can be shown to be minimal as at least one state must be a final state to accept strings ending on a* 1*. Another state must be non-final so that strings ending in a* 0 *are rejected. Thus any FA which accepts the language* $(0|1)^*1$ *must have at least two states.*

As both M_1 *and* M_2 *have exactly two states and both accept the language* $(0|1)^*1$*, they are both minimal FAs and equivalent. This quick example proves that minimal NFAs are not unique.*

Although we are discussing minimal NFAs, it will be interesting to examine some non-minimal NFAs which accept the language $(0|1)^*1$*. For this reason we construct two NFAs,* M_3 *and* M_4 *with three states as follows:*

Let $M_3 = (\{q_0^3, q_1^3, q_2^3\}, \{0,1\}, \delta_3, \{q_0^3\}, \{q_2^3\})$ *with* δ_3 *given by (also shown in figure 2.2)*

	0	1
q_0^3	$\{q_1^3\}$	$\{q_2^3\}$
q_1^3	$\{q_0^3, q_1^3\}$	$\{q_2^3\}$
q_2^3	$\{q_0^3\}$	$\{q_2^3\}$

.

Let $M_4 = (\{q_0^4, q_1^4, q_2^4\}, \{0,1\}, \delta_4, \{q_0^4\}, \{q_2^4\})$ *with* δ_4 *given by (also shown*

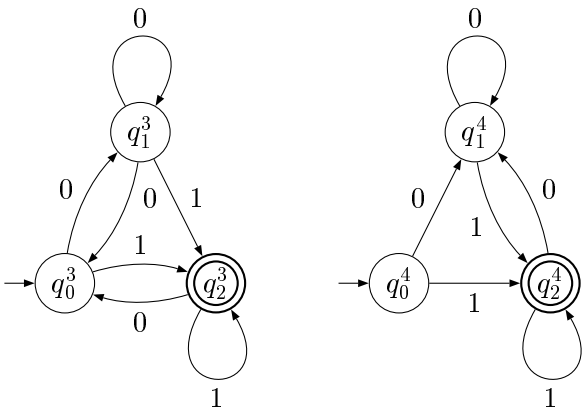

Figure 2.3: Example 2.8. Left: M_3. Right: M_4

in figure 2.2)

	0	1
q_0^4	$\{q_1^4\}$	$\{q_2^4\}$
q_1^4	$\{q_1^4\}$	$\{q_2^4\}$
q_2^4	$\{q_1^4\}$	$\{q_2^4\}$

.

It is easy to verify that $\mathcal{L}(M_3) \equiv \mathcal{L}(M_4) \equiv (0|1)^*1$.

This concludes our discussion on the equivalence and minimality of FAs. In the introduction we mentioned that traditional FAs are divided into two classes, one a simpler form than the other. We discuss the differences in these classes in the next section as well as a method to transform FAs from one class into an equivalent FA of the other.

2.3 Subset Construction

Finite automata are divided into two classes, namely, deterministic and non-deterministic. The main difference between the two classes is that a deterministic finite automaton has at most one state in its active state set while a non-deterministic finite automaton can have more than one state in its active state set. Formally we define a deterministic finite automaton as follows:

Definition 2.9 (Deterministic Finite Automaton)
A FA $M = (Q, \Sigma, \delta, Q_0, F)$ *is a* ***deterministic finite automaton*** *(DFA) if*

$$\begin{aligned} |Q_0| &= 1, \text{ and} \\ |\delta(q, \sigma)| &\leq 1 \quad , \end{aligned}$$

for all $\sigma \in \Sigma$.
Otherwise, M is a ***non-deterministic finite automaton*** *(NFA).*

Note that a DFA is also by default a NFA.

Since $|Q_0| = 1$ in a DFA, the initial state set is commonly denoted as q_0 to show that it only consists of one state which corresponds to the only initial state. The transition table can be written without sets as each transition is either empty or goes to a single state.

Note 2 *A DFA is defined as* ***complete*** *if the inequality in definition 2.9 is an equality. That is, in a complete DFA one state is always active.*

An incomplete DFA can be converted into a complete DFA by simply adding a ***sink state****, s, where any transition $\delta(q, \sigma) = \emptyset$ becomes a transition to the sink state $\delta(q, \sigma) = s$, where $q \in Q$ and $\sigma \in \Sigma$.*

Many minimization algorithms on DFAs only work with complete DFAs. As it is trivial to convert an incomplete DFA to a complete DFA, we henceforth assume that all DFAs are complete.

In the following example we provide an example of a complete DFA and a NFA.

Example 2.10

The FA M from example 2.2 (page 7) has the property that

$$|\delta(q_1, 0)| = |\{q_0, q_1\}| = 2,$$

so that it is a NFA.

The FA M_2 from example 2.8 (page 11) has the property that

$$|\delta(q_i, \sigma)| = 1, \text{ where } \sigma = 0, 1 \text{ and } i = 0, 1$$

so that it is a complete DFA.

There is a well known algorithm, known as *subset construction*, to create an equivalent DFA from any given NFA [15]. This is generated by creating a DFA with states corresponding to each subset of states of the NFA.

The subset construction algorithm is defined as follows:

Definition 2.11 (Subset Construction)

Given a NFA $M = (Q, \Sigma, \delta, Q_0, F)$, let $P = 2^Q$ be the power set[2] of Q. We can now construct an equivalent DFA[3] $M' = (P, \Sigma, \delta', p_0, F')$ where

- $p \in P$ *iff* $p \subseteq Q$,
- $\delta'(p, \sigma) = \bigcup_{q \in p} \delta(q, \sigma)$, $\forall p \in P$ *and* $\forall \sigma \in \Sigma$,
- $p_0 = Q_0$, *and*

[2] See convention A.4, page 170.

[3] The proof for M' being a DFA equivalent to M is known. See for example [15], page 55.

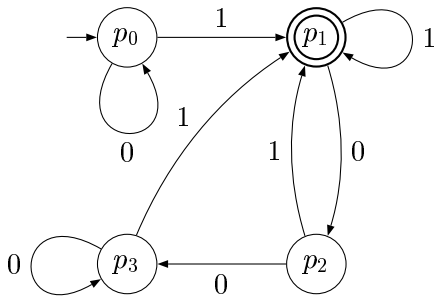

Figure 2.4: Subset constructed DFA for Example 2.2.

- $p \in F'$ *iff* $\exists q \in p$ *such that* $q \in F$.

The above definition of the subset construction creates a DFA with a state corresponding to every subset of the states of the NFA. However, not every subset of states of the NFA is reachable from the initial set of states. The subset construction method can be modified to include only reachable subsets of states by modifying the first requirement above as follows:

- $p \in P$ iff $p \subseteq Q$ and $\exists x \in \Sigma^*$ so that $\delta(q, x) = p$.

Convention 2.12
We use the notation DFA(M) to represent the modified subset construction algorithm on a FA M.

Example 2.13 *Let M be the NFA given in example 2.2 (page 7). Applying the modified definition of subset construction to M we obtain the following DFA:*
$M' = \text{DFA}(M) = (P, \Sigma, \delta', p_0, F')$ *where*

- $P = \{p_0, p_1, p_2, p_3\}$,
- $\Sigma = \{0, 1\}$,
- *the transition function can be written as (also given in figure 2.4)*

p	$\subseteq Q$	0	1
p_0	$\{q_0\}$	p_0	p_1
p_1	$\{q_2\}$	p_2	p_1
p_2	$\{q_1\}$	p_3	p_1
p_3	$\{q_0, q_1\}$	p_3	p_1

, *and*

- $F' = \{p_1\}$.

It is easy to verify that both the DFA M' and the NFA M accept the same language namely $(0|1)^*1$.

This concludes the preliminary chapter for FAs. We have provided a formal definition of FA and differentiated between DFAs and NFAs. We have also discussed the minimality of FAs in terms of equivalence and language as well as providing the subset construction algorithm. These basic concepts will be used in chapter 4 to determine a minimal FA representing a given language. We discuss these algorithms with the goal of using them as a basis for the minimization of symmetric difference FAs, which we define in the following chapter.

In the following chapter we discuss and define the basic concepts of FAs such as equivalence in terms of general FAs and symmetric difference FAs. We will use these concepts with the FA minimization techniques of chapter 4 to attempt to construct minimization algorithms for symmetric difference FAs in chapter 5.

Chapter 3

Generalized FA

Traditional FAs, as defined in chapter 2, have been investigated in the literature since the early 1930s [11]. These traditional FAs have received a large amount of attention both academically and in practice. The majority of applications which make use of traditional FAs use DFAs due to their simplicity. However, NFAs can describe more languages for a given number of states than DFAs with the same number of states. For this reason many of these applications begin by constructing a NFA which they transform into an equivalent DFA for execution.

Recently, a new form of NFAs have been developed, that of generalized non-deterministic FAs ($*$-FAs) [17]. Where traditional NFAs make use of the union rule when calculating the interaction of the transitions, $*$-FAs generalize this rule by replacing the union rule with any binary associative commutative operator. Described as such, traditional NFAs are a special case of $*$-FAs.

It was shown by van Zijl [17] that one such case of these $*$-FAs, namely the symmetric difference FAs ($\oplus$-FAs), are at least as descriptive as traditional NFAs. Therefore, $*$-FAs accept at least as many languages as their traditional NFA counter parts for the same number of states. This implies that some $*$-FAs may be smaller than their equivalent NFAs which will make them of use in applications. Although still a relatively young field, such applications have been developed which make use of $\oplus$-FAs. Examples include hashing [14] and random number generation [18].

In this chapter we provide the basic definitions for $*$-FAs and discuss two cases of these $*$-FAs namely intersection FAs ($\cap$-FAs) and $\oplus$-FAs. The goal of this chapter is to provide a basic understanding of $\oplus$-FAs which can be kept in mind while discussing traditional FA minimization algorithms.

We start by defining $*$-FAs, after which we discuss the properties of these $*$-FAs such as the language, transition function extensions and subset construction. In the second section we discuss the first case of $*$-FAs namely $\cap$-FAs as an example. We focus our discussion of $\cap$-FAs on the execution,

language and subset construction on this case of $\cap$-FAs. In the third section we use the format of our discussion of $\cap$-FAs to discuss the case of $*$-FAs.

3.1 Generalized Finite Automata

$*$-FAs generalize the way in which the transitions from the current active states interact to form the new active state set. In a NFA the active state set is determined by the union of the transitions from the current active states (as given by the extension of the transition function[1]). $*$-FAs define the interaction of the transitions from the current active states on the $*$ operator. The $*$ operator can be any binary associative commutative operator defined on sets[2].

In this section we provide the basic definitions for $\oplus$-FAs which correspond to those for traditional NFAs from the previous chapter. We leave the discussion of these definitions to the examples in the next two sections. With this in mind we define $*$-FAs as follows:

Definition 3.1 (Generalized Finite Automata)
*A **Generalized Finite Automaton** ($*$-FA) M is defined as a 6-tuple $M = (Q, \Sigma, \delta, Q_0, F, *)$ where*

- *Q is the finite, non-empty set of **states**,*
- *Σ is the **alphabet**,*
- *$\delta : Q \times \Sigma \to 2^Q$ is the **transition function**,*
- *Q_0 is the set of **start states**,*
- *F is the set of **final states**, and*
- *$* : 2^Q \times 2^Q \to 2^Q$ is any binary associative commutative operator defined on sets of states.*

A $*$-FA's transition function can be represented graphically or as a table as discussed for FAs in section 2.1 (pages 7, 8).

As with traditional NFAs the transition function of the $*$-FAs can be extended to include sets of states and sequences of alphabet symbols. A $*$-FA's transition function is extended in terms of its operator as shown in the following corollary.

Corollary 3.2 (Extensions)
In a $*$-FA $M = (Q, \Sigma, \delta, Q_0, F, *)$ the domain and range of the transition function can be extended as follows:

[1] See definition 2.5, page 9.

[2] For examples of binary associative commutative operators see definition A.3, page 169.

- $\delta : 2^Q \times \Sigma \to 2^Q$ *with* $S \subseteq Q$ *and* $\sigma \in \Sigma$ *as follows:*

$$\delta(S, \sigma) = \overset{*}{_{q \in S}} \delta(q, \sigma), \text{ and}$$

- $\delta : Q \times \Sigma^* \to 2^Q$ *with* $q \in Q$, $\sigma \in \Sigma$ *and* $x \in \Sigma^*$

$$\begin{array}{lcl} \delta(q, \epsilon) & = & \{q\} \\ \delta(q, \sigma x) & = & \overset{*}{_{q' \in \delta(q,x)}} \delta(q', \sigma) \ . \end{array}$$

From definition 3.1 and corollary 3.2 it is easy to verify that the definition of the $\cup$-FA $M = (Q, \Sigma, \delta, Q_0, F, \cup)$ is equivalent to the definition of the traditional NFA $M = (Q, \Sigma, \delta, Q_0, F)$.

Recall from definition 2.4 (page 9) that the language of a FA is the set of all words accepted by the FA. Similarly, using the extensions discussed in corollary 3.2 we define the language of a $*$-FA as follows:

Definition 3.3 (Language)
The language $\mathcal{L}(M)$ *of a* $*$*-FA* $M = (Q, \Sigma, \delta, Q_0, F, *)$ *is defined as*

$$\mathcal{L}(M) = \{x | x \in \Sigma^*, \ \delta(Q_0, x) \cap F \neq \emptyset\}.$$

We define the equivalence of two $*$-FAs based on the definition of the language of a $*$-FA. This is a generalized format of the definition of equivalence for traditional NFAs 2.6 (page 10).

Definition 3.4 (Equivalence of $*$-FAs)
The $*$*-FA* $M = (Q, \Sigma, \delta, Q_0, F, *)$ *and* $*'$*-FA* $M' = (Q', \Sigma, \delta', Q_0', F', *')$ *are* ***equivalent*** *iff*

$$\mathcal{L}(M) = \mathcal{L}(M').$$

We write $M \equiv M'$ *if* M *is equivalent to* M'.

Note that definition 3.4 does not limit equivalence to a single operator. It follows that two $*$-FAs may be equivalent although they use different operators. If $* = *'$ in definition 3.4 we say that M is equivalent to M' in $*'$-FAs.

Comparison of $*$-FAs using operators such as the $\oplus$ operator is not always easy to follow by hand. For this reason it will be useful to be able to transform such $*$-FAs into a more manageable format. As with the traditional FAs a method exists to construct a DFA equivalent to a given $*$-FA. This method follows the subset construction algorithm from definition 2.11 (page 13) as follows:

Definition 3.5 ($*$-FA Subset Construction)
Given a $*$*-FA* $M = (Q, \Sigma, \delta, Q_0, F, *)$, *let* $P = 2^Q$ *be the power set*[3] *of* Q. *We can now construct an equivalent DFA* $M' = (P, \Sigma, \delta', p_0, F')$ *where*

[3] See convention A.4, page 170.

- $p \in P$ *iff* $p \subseteq Q$,
- $\delta'(p, \sigma) = \overset{*}{_{q \in p}} \delta(q, \sigma)$, $\forall p \in P$ *and* $\forall \sigma \in \Sigma$,
- $p_0 = Q_0$, *and*
- $p \in F'$ *iff* $\exists q \in p$ *such that* $q \in F$.

On-the-fly subset construction for traditional NFAs constructs DFAs without non reachable states from the initial state set (page 14). Similarly the $*$-FA subset construction algorithm can be modified to exclude nonreachable states from the initial state set.

Note 3

We define the $*$-FA to construct DFAs and not deterministic $*$-FAs. We do this to keep discussion of the algorithm simple. However, as a DFA can only ever have one state in the active state set and as the operator of a $*$-FAs is commutative we have that $\delta(q, \sigma) = \delta_*(q, \sigma)$ for any state q and alphabet symbol σ from a FA. It follows that all deterministic $*$-FAs are DFAs.

In this section we have defined the properties of and subset construction algorithm on $*$-FAs. Before discussing symmetric difference FAs we discuss another form of $*$-FAs, namely the intersection FA. We provide examples for the concepts described above in terms of the intersection FAs and how they relate to the traditional FAs discussed in chapter 2. We use the examples we discuss for intersection FAs as a basis for our discussion of symmetric difference FAs.

3.2 Intersection FAs

In this section we discuss examples of the $*$-FA's concepts we defined above on intersection FAs. Although this thesis does not focus on intersection FAs we use it as a basis for approaching $\oplus$-FAs. We will also show how some algorithms we will discuss for symmetric difference FAs can be adapted for $\cap$-FAs.

Intersection FAs are a form of the $*$-FAs using the intersection rule[4], defined as follows:

Definition 3.6 (Intersection FA)

A $*$-FA $M = (Q, \Sigma.\delta, Q_0, F, *)$ is defined as an intersection FA ($\cap$-FA) if $* = \cap$, that is

$$M = (Q, \Sigma, \delta, Q_0, F, \cap).$$

We discuss examples to highlight the following differences between $\cap$-FA and traditional NFAs:

[4]See definition A.3, page 169.

1. Active state sets,
2. languages, and
3. subset construction.

Example 3.7 (Active State Sets)
For use in the examples we construct the $\cap$-FA $M_\cap = (Q, \Sigma, \delta, Q_0, F, \cap)$ and $M_\cup = (Q, \Sigma, \delta, Q_0, F, \cup)$ from the NFA M from example 2.2 (page 7). That is,

- $Q = \{q_0, q_1, q_2\}$,
- $\Sigma = \{0, 1\}$,
- δ *can be described as (also shown in figure 3.1)*

	0	1
q_0	$\{q_0\}$	$\{q_2\}$
q_1	$\{q_0, q_1\}$	$\{q_2\}$
q_2	$\{q_1\}$	$\{q_2\}$

,

- $Q_0 = \{q_0\}$, *and*
- $F = \{q_2\}$.

Executing both $M_\cap$ and $M_\cup$ on the input 01000 *we have the following series of active state sets:*

δ	σ	$M_\cap$	$M_\cup$
$\{q_0\}$	0	$\{q_0\}$	$\{q_0\}$
$\{q_0\}$	1	$\{q_2\}$	$\{q_2\}$
$\{q_2\}$	0	$\{q_1\}$	$\{q_1\}$
$\{q_1\}$	0	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	0	$\{q_0\}$	$\{q_0, q_1\}$

.

From the series of active state sets we have that in $M_\cap$,

$$\delta(\{q_0, q_1\}, 0) \;=\; \delta(q_0, 0) \cap \delta(q_1, 0) \;=\; \{q_0\} \cap \{q_0, q_1\} \;=\; \{q_0\},$$

while in $M_\cup$,

$$\delta(\{q_0, q_1\}, 0) \;=\; \delta(q_0, 0) \cap \delta(q_1, 0) \;=\; \{q_0\} \cup \{q_0, q_1\} \;=\; \{q_0, q_1\},$$

so that the transition differs between $M_\cap$ and $M_\cup$ from the set $\{q_0, q_1\}$ on the alphabet symbol 0.

We expand on this singular case of differing transitions in the previous example to discuss the difference in the languages accepted by $M_\cap$ and $M_\cup$.

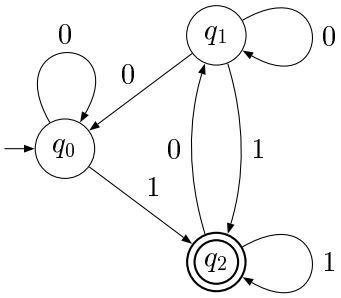

Figure 3.1: Example 3.7.

Example 3.8 (Languages)

Let $M_\cap$ and $M_\cup$ be the FAs from example 3.7. As $M_\cup$ is equivalent to the FA M from example 2.2 (page 7), we have that $\mathcal{L}(M) = (0|1)^*1$ (page 10).

For the $\cap$-FA $M_\cap$ we determine the language of the $\cap$-FA by examining the possible active state sets and their resultant active state sets:

	0	1
$\{q_0\}$	$\{q_0\}$	$\{q_2\}$
$\{q_2\}$	$\{q_1\}$	$\{q_2\}$
$\{q_1\}$	$\{q_0, q_1\}$	$\{q_2\}$
$\{q_0, q_1\}$	$\{q_0\}$	$\{q_2\}$

.

For any state set S from the above set of active state sets, we have that $\delta(S, 0) \cap F = \emptyset$ and $\delta(S, 1) \cap F \neq \emptyset$. This implies that the language accepted by $M_\cap$ is $(0|1)^*1$ so that

$$\mathcal{L}(M_\cap) \;=\; \mathcal{L}(M_\cup).$$

In example 3.8 $M_\cup$ and $M_\cap$ are equivalent, even though they use different operators. It is important not to assume this is true in the general case.

In the final example we examine the subset construction transformation on $M_\cup$ and $M_\cap$ from the examples.

Example 3.9 (Subset Construction)

Let $M_\cap$ and $M_\cup$ be the FAs from example 3.7. Furthermore, let $M'_\cup = \mathrm{DFA}(M_\cup) = (P, \Sigma, \delta'_\cup, p_0, F'_\cup)$ and $M'_\cap = \mathrm{DFA}(M_\cap) = (P', \Sigma, \delta'_\cap, p'_0, F'_\cap)$ determined as follows.

As $M_\cup = M$ from example 2.2 (page 7), we have that $\mathrm{DFA}(M_\cup) = M'$ from example 2.13 (page 14), so that $M'_\cup = (P, \Sigma, \delta'_\cup, p_0, F'_\cup)$ with

- $P = \{p_0, p_1, p_2, p_3\}$,

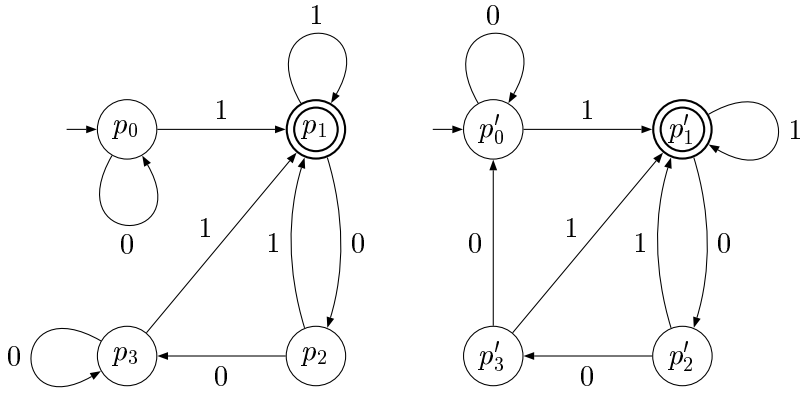

Figure 3.2: Example 3.9. Left $M'_{\cup}$. Right $M'_{\cap}$.

- $\Sigma = \{0, 1\}$,
- *the transition function can be written as (also given in figure 3.2, left)*

$\delta'_{\cup}$	$\subseteq Q$	0	1
p_0	$\{q_0\}$	p_0	p_1
p_1	$\{q_2\}$	p_2	p_1
p_2	$\{q_1\}$	p_3	p_1
p_3	$\{q_0, q_1\}$	p_3	p_1

, and

- $F'_{\cup} = \{p_1\}$.

Using definition 3.5 and the $\cap$-FA $M_{\cap}$ from example 3.8 we can construct the DFA $M'_{\cap}$ with $M'_{\cap} = (P', \Sigma, \delta'_{\cap}, p'_0, F'_{\cap})$ where

- $P' = \{p'_0, p'_1, p'_2, p'_3\}$,
- $\Sigma = \{0, 1\}$,
- *the transition function can be written as (also given in figure 3.2, right.)*

$\delta'_{\cap}$	$\subseteq Q$	0	1
p'_0	$\{q_0\}$	p'_0	p'_1
p'_1	$\{q_2\}$	p'_2	p'_1
p'_2	$\{q_1\}$	p'_3	p'_1
p'_3	$\{q_0, q_1\}$	p'_0	p'_1

, and

- $F'_{\cap} = \{p'_1\}$.

In example 3.9 it is of interest to note that even though $M_{\cup} \equiv M_{\cap}$ and they share the same transition function, the resultant DFA through subset

construction differ. The reason for this is the resultant active state sets due to the operator used.

This concludes the examples and discussion of the $\cap$-FAs. We follow on with the format used in this section to discuss $\oplus$-FAs.

3.3 Symmetric Difference FAs

In the previous section we discussed a case of the $*$-FA, namely $\cap$-FA. In this section we follow the format used in the previous section to define and discuss symmetric difference FAs. After formally defining $\oplus$-FAs we discuss three examples of the properties of symmetric difference FAs, namely the execution, language and subset construction. Once again we compare the examples with the respective traditional FAs properties.

The symmetric difference operator is denoted as the symbol $\oplus$ and is defined in definition A.3 (page 169).

We define a symmetric difference FA as follows:

Definition 3.10 (Symmetric Difference FA)
A $*$-FA $M = (Q, \Sigma.\delta, Q_0, F, *)$ is defined as a symmetric difference FA ($\oplus$-FA) if $* = \oplus$, that is

$$M = (Q, \Sigma, \delta, Q_0, F, \oplus).$$

As with $\cap$-FA the first example of the $\oplus$-FA we discuss deals with the execution of the $\oplus$-FA.

Example 3.11 (Active State Sets)
For use in the examples we construct the $\oplus$-FA $M_\oplus = (Q, \Sigma, \delta, Q_0, F, \oplus)$ and $M_\cup = (Q, \Sigma, \delta, Q_0, F, \cup)$ from the NFA M from example 2.2 (page 7). That is,

- $Q = \{q_0, q_1, q_2\}$,
- $\Sigma = \{0, 1\}$,
- *δ can be described as (also shown in figure 3.3)*

	0	1
q_0	$\{q_0\}$	$\{q_2\}$
q_1	$\{q_0, q_1\}$	$\{q_2\}$
q_2	$\{q_1\}$	$\{q_2\}$

,

- $Q_0 = \{q_0\}$, *and*
- $F = \{q_2\}$.

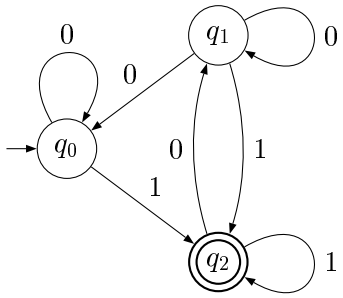

Figure 3.3: Example 3.11.

Executing both $M_\oplus$ *and* $M_\cup$ *on the input* 01000 *we have the following series of active state sets:*

δ	σ	$M_\oplus$	$M_\cup$
$\{q_0\}$	0	$\{q_0\}$	$\{q_0\}$
$\{q_0\}$	1	$\{q_2\}$	$\{q_2\}$
$\{q_2\}$	0	$\{q_1\}$	$\{q_1\}$
$\{q_1\}$	0	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	0	$\{q_1\}$	$\{q_0, q_1\}$

.

From the series of active state sets we have that in $M_\oplus$*,*

$$\delta(\{q_0, q_1\}, 0) \;=\; \delta(q_0, 0) \oplus \delta(q_1, 0) \;=\; \{q_0\} \oplus \{q_0, q_1\} \;=\; \{q_1\},$$

while in $M_\cup$*,*

$$\delta(\{q_0, q_1\}, 0) \;=\; \delta(q_0, 0) \oplus \delta(q_1, 0) \;=\; \{q_0\} \cup \{q_0, q_1\} \;=\; \{q_0, q_1\},$$

so that the transition differs between $M_\oplus$ *and* $M_\cup$ *from the set* $\{q_0, q_1\}$ *on the alphabet symbol* 0*.*

As with the $\cap$-FA the $\oplus$-FAs transitions only differ from the state set $\{q_0, q_1\}$ on the alphabet symbol 0. It is also easy to verify by comparing examples 3.7 and 3.11 that $M_\oplus$ and $M_\cap$ also differ only on that transition.

Once again we expand on this singular case of differing transitions in the previous example to discuss the difference in the languages accepted by $M_\oplus$ and $M_\cup$.

Example 3.12 (Languages)

Let $M_\oplus$ *and* $M_\cup$ *be the FAs from example 3.11. As* $M_\cup$ *is equivalent to the FA* M *from example 2.2 (page 7), we have that* $\mathcal{L}(M) = (0|1)^*1$ *(page 10).*

For the $\oplus$-FA $M_\oplus$ we examine the possible active state sets and their following active state sets:

	0	1
$\{q_0\}$	$\{q_0\}$	$\{q_2\}$
$\{q_2\}$	$\{q_1\}$	$\{q_2\}$
$\{q_1\}$	$\{q_0, q_1\}$	$\{q_2\}$
$\{q_0, q_1\}$	$\{q_1\}$	$\{q_2\}$

.

For any state set S from the above set of active state sets, we have that $\delta(S, 0) \cap F = \emptyset$ and $\delta(S, 1) \cap F \neq \emptyset$. Therefore, the language accepted by $M_\oplus$ is $(0|1)^*1$ so that

$$\mathcal{L}(M_\oplus) \;=\; \mathcal{L}(M_\cup).$$

In example 3.12 we have that as with $M_\cap$ from example 3.8 (page 21), $M_\oplus$ is equivalent to $M_\cup$. This in turn implies that $M_\oplus \equiv M_\cap$ from the two examples. Once again this is not the general case but works well for highlighting the examples. As with $M_\cap$ we will see that DFA($M_\oplus$) results in a different but equivalent DFA to both $M_\cup$ and $M_\cap$.

Example 3.13 (Subset Construction)

Let $M_\oplus$ and $M_\cup$ be the FAs from example 3.11. Furthermore, let $M'_\cup =$ DFA($M_\cup$) $= (P, \Sigma, \delta'_\cup, p_0, F'_\cup)$ and $M'_\oplus =$ DFA($M_\oplus$) $= (P', \Sigma, \delta_c ap', p'_0, F'_\oplus)$ as follows.

As $M_\cup = M$ from example 2.2 (page 7), we have that $M_\cup = M'$ from example 2.13 (page 14), so that $M'_\cup = (P, \Sigma, \delta'_\cup, p_0, F'_\cup)$ with

- $P = \{p_0, p_1, p_2, p_3\}$,
- $\Sigma = \{0, 1\}$,
- *the transition function can be written as (also given in figure 3.2, left)*

$\delta'_\cup$	$\subseteq Q$	0	1
p_0	$\{q_0\}$	p_0	p_1
p_1	$\{q_2\}$	p_2	p_1
p_2	$\{q_1\}$	p_3	p_1
p_3	$\{q_0, q_1\}$	p_3	p_1

, and

- $F'_\cup = \{p_1\}$.

Using definition 3.5 and example 3.9 we construct the DFA $M'_\oplus$ with $M'_\oplus = (P', \Sigma, \delta'_\oplus, p'_0, F'_\oplus)$ where

- $P' = \{p'_0, p'_1, p'_2, p'_3\}$,
- $\Sigma = \{0, 1\}$,

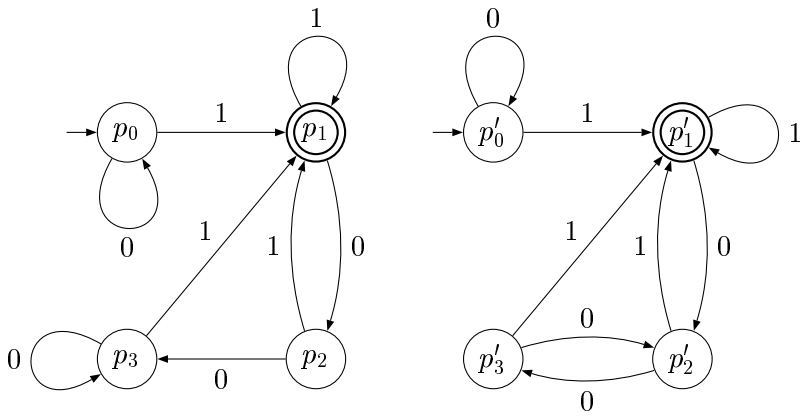

Figure 3.4: Example 3.13. Left $M'_{\cup}$. Right $M'_{\oplus}$.

- *the transition function can be written as (also given in figure 3.4, right)*

$\delta'_{\oplus}$	$\subseteq Q$	0	1
p'_0	$\{q_0\}$	p'_0	p'_1
p'_1	$\{q_2\}$	p'_2	p'_1
p'_2	$\{q_1\}$	p'_3	p'_1
p'_3	$\{q_0, q_1\}$	p'_2	p'_1

, *and*

- $F'_{\oplus} = \{p'_1\}$.

As shown in the example the DFA resultant from $\oplus$-FA is not identical to either the DFA constructed from the $\cup$-FA or $\cap$-FA from example 3.9.

With this example we finish our introduction to $\oplus$-FAs. We have provided the basic definitions of $\oplus$-FAs and discussed examples which highlighted differences in the execution and properties of $\oplus$-FAs to $\cup$-FAs. We provided this introduction of $\oplus$-FAs at this stage so they can be kept in mind when discussing the traditional FA minimization algorithms in chapter 4. In chapter 5 we attempt to modify the algorithms from chapter 4 to use with $\oplus$-FAs as defined here.

Before we begin our discussion of traditional FA minimization algorithms we discuss one further property of $*$-FAs.

This concludes our discussion of $*$-FAs and their properties. In this chapter we provided the basic definitions of $*$-FAs with reference to NFAs and gave examples for both the $\cap$-FAs and $\oplus$-FAs. We discuss further properties of $\oplus$-FAs which will be of use for minimization purposes after discussing minimization algorithms on NFAs and DFAs.

In the next chapter we discuss traditional FA minimization algorithms. In the following chapter we use the properties of traditional FAs and their minimization algorithms and attempt to adapt them for use with $\oplus$-FAs as discussed in this chapter.

Chapter 4

Minimization of FA

In chapter 2 we presented definitions and concepts pertaining to FAs. We defined the equivalence of FAs based on their languages and elaborated that a FA was minimal if no equivalent FA exists with fewer states. In this chapter we discuss various techniques which exist in the literature to determine minimal FAs.

Various applications make use of large FAs which consist of hundreds of thousands of states. Examples of such applications include linguistics, language processing, pattern searching and computational biology [4]. Although, non minimal FAs could be used in such applications, a minimal FA decreases the storage requirements and improves the efficiency of such applications. Among other improvements a minimal FA reduces the impact of state explosion[1].

As shown in [19] the minimization of FAs is a well researched topic with various algorithms available. These FA minimization algorithms are divided into two types, namely DFA and NFA minimization algorithms. As DFAs are an older concept and have a simpler structure than NFAs, most FA minimization algorithms deal exclusively with DFAs[2].

DFA and NFA minimization algorithms offer various advantages over each other. Due to the simpler structure, DFA minimization algorithms are faster and simpler than those for NFAs and result in the unique equivalent minimal DFA. Similarly, NFA minimization algorithms are usually slower than their DFA counterparts and the resultant minimal NFA is not unique. However, minimal NFAs always consist of fewer or equal number of states the equivalent minimal DFA. In practice more applications make use of DFA minimization due to the worst case computational complexity of NFA minimization algorithms. Even in such applications an efficient NFA reduction algorithm could improve the performance of the DFA minimization if the

[1] Various transformations on FAs, including subset construction, have exponential worst case computational complexities and/or create FAs which consist of a state set of exponential size compared to the original FAs state set.

[2] DFA minimization algorithms date back to as early as the 1960's [2].

original FA is an NFA. This is due to the space explosion inherent to the subset construction method used to create equivalent DFAs from NFAs. It is important to note that as shown by Jiang and Ravikumar [10] NFA minimization is a NP-Hard problem and an efficient NFA minimization algorithm is unlikely to be found.

Recently, some attention has been given to reduction of NFAs instead of minimization. In this approach, techniques are used to reduce the number of states and/or transitions of the NFA, without necessarily achieving the lowest possible (i.e. minimal) number of states. Ilie and Yu [9] and Champarnaud and Coulon [3] developed such reduction techniques that run in polynomial time. This is a considerable improvement of worst case computational complexity over NFA minimization algorithms and as described above can be used to improve the efficiency of applications making use of DFA minimization algorithms.

Although we do not discuss this in detail, it is of interest note that as the storage of FAs depends not only on the number of states, but also the number of transitions, the minimization and reduction of transitions has become a new area of research. The algorithm we discuss later by Ilie and Yu [9] reduces not only the number of states, but also the number of transitions.

We start our discussion of the reduction and minimization of $\oplus$-FAs in terms of the number of states by discussing reducible states and types of minimization algorithms. Using these concepts we discuss three DFA minimization algorithms in section 4.2, namely dictionary minimization [4], Hopcroft's [7] and Brzozowksi's [2] DFA minimization algorithms. Using these algorithms as a basis we discuss the NFA minimization by Kameda and Weiner [11] and the NFA reduction algorithm by Ilie and Yu [9].

We make use of these concepts and algorithms as a basis for $\oplus$-FA minimization in chapter 5.

4.1 Preliminaries

Although a wide range of minimization algorithms are available in the literature, we focus our discussion on the following three categories of the minimization algorithms:

- **Language construction**: The language that the FA accepts is used to construct a minimal FA which accepts the language;
- **State comparison**: States are combined or removed from the FA reducing the number of states in the FA without affecting the language it accepts; and
- **Transformation construction**: Transformations are applied to the FA resulting in an equivalent FA with fewer states.

In the case of language construction we present a DFA minimization algorithm. However, in the cases of the state comparison and transformation construction algorithms we present minimization algorithms for both DFAs and NFAs. The DFA minimization algorithms we discuss are those of Daciuk, Watson and Watson [4] for language reconstruction, Hopcroft [7] for state comparison and Brzozowski [2] for transformation reconstruction. The NFA minimization and reduction algorithms we discuss are those of Ilie, Navarro and Yu [9] for state comparison and Kameda and Weiner [11] for transformation construction.

The minimization algorithms we discuss are based on the concept of reducible states. Reducible states are states which can be removed from a FA or be used to transform a FA in such a way that an equivalent FA can be constructed with fewer states. Before discussing the minimization and reduction algorithms we provide a formal definition of these reducible states and discuss how they can be detected.

Reducible states can formally be defined as follows:

Definition 4.1 (Reducible states)
Given the FA $M = (Q, \Sigma, \delta, Q_0, F)$*, the* ***reducible states*** *are defined as follows:*

- q *is reducible if*

  $$\mathcal{L}(M) \equiv \mathcal{L}((Q/\{q\}, \Sigma, \delta', Q_0/\{q\}, F/\{q\}))$$

  where $\delta'(q', \sigma) = \delta(q', \sigma)/\{q\}$ *with* $q' \in Q/\{q\}$*, or*

- q_i*,* q_j *are reducible if for a* $Q' = (Q/\{q_i,\ q_j\}) \cup \{q'\}$ *and a* $\delta' : Q' \times \Sigma \to 2^{Q'}$ *it holds that*

  $$\mathcal{L}(M) \equiv \mathcal{L}((Q', \Sigma, \delta', Q_0', F')),$$

  where $Q_0' = Q_0$ $(F' = F)$*, if* $q_i, q_j \notin Q_0$ $(q_i, q_j \notin F)$*,*
  otherwise

  $$Q_0' = (Q_0/\{q_i, q_j\}) \cup \{q'\}\ (F' = (F/\{q_i, q_j\}) \cup \{q'\}).$$

The first case in definition 4.1 is that of a state as being reducible if

- it can be removed from the state set,
- it can be removed from the initial and final state sets, and
- all transitions to and from the state are removed,

without affecting the language accepted by the FA.

The second case in definition 4.1 is that of a pair of states as being reducible if

- they can be removed from the state set and replaced with a single replacement state,
- they can be replaced with the replacement state in the initial and final state sets, and
- all transitions to and from the state are replaced with transitions to and from the replacement state,

without affecting the language accepted by the FA.

The definition provides us with a method to remove the reducible states from the FA to create a smaller equivalent FA. However, it does not present a method to determine which states are reducible. We could use brute force methods to determine which states are reducible, but such a brute force method would have a high worst case computational cost associated.

To avoid this worst case computational complexity we look into concepts which will simplify the process of identifying reducible states. The first such concept we discuss originates from [11] and deals with the words which occur before and after a given state.

Definition 4.2 (Predecessor/Successor)
Given the FA $M = (Q, \Sigma, \delta, Q_0, F)$ with a state $q \in Q$:

- *the **predecessor**[3] of the state q is the set of all the words which, when run on M with the original active state set $= Q_0$, result in the state q being active:*

$$pr(q) = \{x \mid x \in \Sigma^*,\ q \in \delta(Q_0, x)\},\ \textit{and}$$

- *the **successor** of the state q is the set of all the words which, when run on M with the original active state set $= \{q\}$, results in a final state of M being active:*

$$sc(q) = \{x \mid x \in \Sigma^*,\ \delta(q, x) \bigcap F \neq \emptyset\}.$$

From definition 4.2 we see that the definition of the predecessor (resp. successor) if used on the final (resp. initial) state set is equivalent to the language of the FA, that is:

$$\begin{aligned} pr(q) &= \mathcal{L}((Q, \Sigma, \delta, Q_0, q)) \\ sc(q) &= \mathcal{L}((Q, \Sigma, \delta, q, F)) \end{aligned}$$

It follows that:

[3]Note the difference between the definitions of successors (resp. predecessors) and the children (resp. parents) of a state.

Corollary 4.3 (Complexity)
The complexity of calculating the successor or predecessor of a state in a FA is as difficult as calculating the language of the FA.

The predecessors and successors of the states are used to examine whether or not states are necessary to generate the words in the language of the FA. For example if the predecessor and successor of a state form a subset of another states predecessor and successor then any word in the language of the FA which sets the first state active will set the second state active at the same time. In such a case the state is repeating work done by the other state and the first state can be removed without affecting the language of the FA. We focus on four cases of reducible states identifiable using predecessors and successors.

From definition 4.1 (page 29) we can identify the following types of reducible states based on the predecessors and successors of the states, as shown in [11] and [19].

Definition 4.4 (Reducible States)
In a FA $M = (Q, \Sigma, \delta, Q_0, F)$ *with a state* $q \in Q$

- q *is* ***initial-useless*** *if*
$$pr(q) = \emptyset,$$
- q *is* ***final-useless*** *if*
$$sc(q) = \emptyset,$$
- q *is* ***redundant*** *with respect to another state* $q' \in Q$ *if*
$$pr(q) = pr(q'), \text{ and}$$
- q *is* ***equivalent*** *to another state* $q' \in Q$ *if*
$$sc(q) = sc(q').$$

From definition 4.4 we have that a state is initial-useless if its predecessor is the empty set. Similarly, a state is final-useless if its successor is the empty set. Where a state's predecessor is identical to another state's predecessor we define the state as redundant with respect to the other state. Similarly, where a state's successor is identical to another state's successor we define the state as equivalent to the other state.

In the following lemmas we provide proofs that initial-useless, final-useless and redundant states are reducible. These proofs are of interest as they provide algorithms to remove these reducible states from a FA.

Lemma 4.5 *Initial-useless and final-useless states are reducible.*

Proof We must show that by removing the initial-useless and final-useless states from a FA M we can generate an equivalent FA M' with fewer states.

Let $M = (Q, \Sigma, \delta, Q_0, F)$ be a FA. From definition 2.4 (page 9), we have that the language of M is given by

$$\mathcal{L}(M) = \{x | x \in \Sigma^*,\ \delta(Q_0, x) \bigcap F \neq \emptyset\}.$$

Let U be the set of initial-useless and final-useless states in M. Let the FA $M' = (Q/U, \Sigma, \delta', Q_0/U, F/U)$ where $\delta'(q, \sigma) = \delta(q, \sigma)/U$ for $q \in Q/U$ and $\sigma \in \Sigma$.

We must show that $\mathcal{L}(M') = \mathcal{L}(M)$. We do this by showing that any word accepted by the FA M does not go through an initial-useless or final-useless state. By that we mean the state sequence which is set active by processing any word in the language of the FA does not need to contain any initial-useless or final-useless state.

Let x be any word $x \in \mathcal{L}(M)$ where $x = yz$ with $y, z \in \Sigma^*$ and $q \in \delta(Q_0, y)$. Then q cannot be initial-useless as $y \in pr(q)$ so that $pr(q) \neq \emptyset$. If q is final-useless, then we have that $sc(q) = \emptyset$. Thus with $A \subseteq Q$ and $q \in A$ we have that $sc(A/\{q\}) = sc(A)$ so that $\delta(\delta(Q_0, y)/\{q\}, z) \bigcap F \neq \emptyset$ if and only if $\delta(\delta(Q_0, y)/\{q\}, z) \bigcap F \neq \emptyset$.

It follows that $\delta'(\delta'(Q_0/U), y), z) \bigcap F/U) \neq \emptyset$ if and only if

$$\delta(\delta(Q_0), y), z) \bigcap F) \neq \emptyset,$$

so that $\mathcal{L}(M) = \mathcal{L}(M')$. □

From the proof we have that FA with initial-useless and final-useless states can be reduced by removing these states and any transitions to or from them from the FA. This corresponds with the first case of reducible states from definition 4.1

Next we show that redundant states are reducible.

Lemma 4.6 *Redundant states are reducible.*

Proof We must show that by removing redundant states from a FA, M, we can generate an equivalent FA, M', with fewer states.

Let $M = (Q, \Sigma, \delta, Q_0, F)$ be a FA where the states q and $p \in Q$ are redundant. We construct the FA $M' = (Q', \Sigma, \delta', Q_0', F')$ where

- $Q' = Q/\{p\}$,
- $Q_0' = Q_0/\{p\}$,
- if $p \in F$ then $F' = (F/\{p\}) \bigcup \{q\}$, and
- δ' for $\sigma \in \Sigma$ is modified from δ as follows

$$\delta'(q_i, x) = \begin{cases} \delta(q_i, \sigma)/\{q\} & ,\ q_i \neq q, p \\ \delta(q, x) \bigcup \delta(p, \sigma)/\{q\} & ,\ q_i = p \end{cases}$$

We construct M' from M as follows:

- Remove the state p from both the state set and initial state set.
- If the state p is a final state in M then we declare the state q as a final state in M'.
- The transition function of M' is constructed from the transition function of M by removing all transitions to the state p and combining all the transitions from the state q with the transitions from the state p.

We need to show that $\mathcal{L}(M') = \mathcal{L}(M)$ or that $\delta(Q_0, x) \bigcap F \neq \emptyset$ if and only if $\delta'(Q'_0, x) \bigcap F' \neq \emptyset$ with $x \in \Sigma^*$. As p is redundant with respect to q we have that $q \in \delta(Q_0, x)$ if and only if $p \in \delta(Q_0, x)$.

Let $x = yz$ where $y, z \in \Sigma^*$ then if $\{q, p\} \subseteq \delta(Q_0, y)$ we have that $\delta(Q_0, y) = \delta'(Q'_0, y) \cup \{p\}$. Let $\delta(Q_0, y) = Q_y = \delta'(Q'_0, y) \cup \{p\}$ and $\delta'(Q'_0, y) = Q'_y$. Where $\sigma \in \Sigma$ we have that

$$\begin{aligned} \delta'(Q'_y, \sigma) &= \delta'(Q'_y/\{q\}, \sigma) \cup \delta'(q, \sigma) \\ &= \delta(Q_y/\{q, p\}, \sigma) \cup \delta'(q, \sigma) \\ &= \delta(Q_y/\{q, p\}, \sigma) \cup \delta(\{q, p\}, \sigma) \\ &= \delta(Q_y, \sigma). \end{aligned}$$

It follows that $\delta'(Q'_y, y\sigma) = \delta(Q_y, y\sigma)$.

Next we take the case where $x \in \mathcal{L}(M)$ with $\{q, p\} \subseteq \delta(Q_0, x) = \delta'(Q'_0, x) \cup \{p\}$. If $q \in F$ then we have that $p \in F'$ so that $\delta'(Q'_0, x) \cap F' \neq \emptyset$. If $q \notin F$ then we have that $(\delta(Q_0, x)/\{q\}) \cap F \neq \emptyset$. As $\delta'(Q'_0, x) = (\delta(Q_0, x)/\{q\})$ it follows that $\delta'(Q'_0, x) \cap F' \neq \emptyset$.

Thus for any word $x \in \mathcal{L}(M)$ with $x = yz$ where $y, z \in \Sigma^*$ and $\sigma \in \Sigma$ we have shown that

- if $q \in \delta(Q_0, y)$ then $\delta(Q_0, y\sigma) = \delta'(Q'_0, y\sigma)$, and
- if $\delta(Q_0, x) \cap F\emptyset$ then $\delta'(Q'_0, x) \cap F' \neq \emptyset$.

It follows that $\mathcal{L}(M) = \mathcal{L}(M')$. □

From the proof we have that redundant states can be reduced by removing one of the redundant states, all transitions to the redundant state and adding all of the transitions from the removed state to the other remaining state. This follows the second case of reducible states from definition 4.1.

We do not provide a proof for the reducibility of equivalent states at this time. The reason for this is due to property that equivalent states are redundant states of the dual FA which we will discuss later. As shown in [19] equivalent states may be reduced by removing one of the states and moving all transitions to the removed state, to the other equivalent state.

Using the concepts we have discussed in this section we will approach the minimization of DFAs. We plan to use these DFA minimization algorithms as a starting point to approach NFA minimization algorithms and in turn $\oplus$-FA minimization algorithms.

4.2 DFA Minimization

DFAs have received the most attention with regards to minimization and as such there exist a plethora of DFA minimization algorithms in the literature. Watson [19] wrote a taxonomy of DFA minimization techniques which standardized the notations of the algorithms and organized them into a family tree of their influences. This taxonomy helps distinguish the different algorithms, however, it does not include the minimization techniques for specialized DFAs such as dictionaries.

In this section we start by discussing properties of DFAs not shared by NFAs. These properties allow the minimization techniques constructed for DFAs to be simpler and faster than those for NFAs. After these properties we discuss three different approaches to minimizing DFAs. The first by Daciuk, Watson and Watson [4] builds a minimal specialized form of a DFA called a dictionary through language construction. The second algorithm by Hopcroft [7] is a form of the most popular and fastest DFA minimization algorithm through state comparison. The final algorithm we discuss is by Brzozowski [2] which makes use of transformation construction. Although this algorithm does not have the speed of the previous two algorithms, it does have some interesting properties which are usable for $\oplus$-FAs. After this work we discuss various properties of the minimal DFA.

Before discussing the minimization techniques for DFAs we look at the reducible states of DFAs.

From the definition of DFAs (see page 12) we have that the DFA has exactly one start state and may never have more than one state in the active state set. The following theorem follows from this:

Theorem 4.7 (Redundancy)
A DFA has no redundant states.

Proof

Let M be a DFA $M = (Q, \Sigma, \delta, Q_0, F)$ with two states $q_i, q_j \in Q$ redundant with respect to each other where $q_i \neq q_j$. From the definition of redundancy we have $pr(q_i) = pr(q_j)$ With $x \in \Sigma^*$ it follows that $q_i \in \delta(Q_0, x)$ if and only if $q_j \in \delta(Q_0, x)$ so that $\{q_i, q_j\} \subseteq \delta(Q_0, x)$.

From definition 2.9 (page 12) we have that $|\delta(q, \sigma)| \leq 1$ wher $x \in \Sigma$. It follows that $|\delta(Q_0, x)| \leq 1$ which forms a contradiction with $\{q_i, q_j\} \subseteq \delta(Q_0, x)$.

By contradiction we have that a DFA has no redundant states. □

We can further reduce the types of reducible states of a DFA by constructing a DFA using the modified subset construction algorithm (page 13). The DFA will then have no initial-useless states.

This means that for the minimization of DFAs we need only handle equivalent and final-useless states. This helps to simplify the minimization problem as we need only identify and reduce two types of reducible states.

We start by discussing the dictionary minimization problem. This problem makes use of a simpler form of the DFA. As such it simplifies the minimization problem further and allows some insight into detecting and removing reducible states.

4.2.1 Dictionaries

A dictionary [4] as used here is a specialized form of a DFA used to store words of finite length, such as English words or DNA sequences. As such, dictionaries are often generated piecewise by inserting each accepted word into the dictionary. Many dictionaries are also interactive, that is, words can be added after the initial construction was completed.

Dictionaries are used in many applications which make use of word matching. These applications include spell checkers and pattern recognition. Dictionaries are also used to store large amounts of distinct data such as DNA sequences in a compact space without data loss.

Three algorithms of dictionary minimization are provided by Daciuk, Watson and Watson [4]. The first algorithm deals with minimizing already constructed dictionaries. The second algorithm constructs the minimal dictionaries from a set of words in alphabetical order. The final algorithm constructs the minimal dictionary from a set of words in no particular order. The final algorithm allows for words to be added to the dictionary after the initial construction is complete.

We do not discuss the last two minimization algorithms in detail, but instead describe the approaches used to these algorithms based on the first minimization algorithm.

Before discussing the minimization algorithms, we start by formally defining and discussing dictionaries.

Definition 4.8 (Dictionary)
*A **dictionary** is a DFA $M = (Q, \Sigma, \delta, Q_0, F)$, with no **loops**, that is,*

$$\forall x \in \Sigma^*,\ q \in Q,\ \textit{it holds that } q \notin \delta(q, x).$$

This means that specialized forms of minimization are needed to ensure that loops do not occur. We start by discussing the structure of dictionaries namely the **TRIE**, after which we discuss the three dictionary minimization algorithms.

TRIEs

TRIEs are the most commonly used data structure to store words for pattern recognition such as dictionaries and search engines[4]. They are tree-structured DFAs in which the initial state is the root and the set of final states consists of all the leaves as well as internal states which mark words that form prefixes to other words in the dictionary.

A TRIE $(Q, \Sigma, \delta, q_0, F)$ can be constructed incrementally. This is done by processing each word separately and adding states and transitions to the structure as needed. Each character of the word is processed until the longest prefix of the word currently stored in the TRIE is found. States are added to the TRIE which represent the remaining suffix.

We define this process formally as follows:

1. Construct the initial and final state sets as $Q = \{q_0\}$, and $F = \{\}$
2. Construct the alphabet set Σ to include the appropriate symbols.
3. **Repeat** until all words have been inserted
   (a) $w \leftarrow$ next word to be inserted,
   (b) $\alpha \leftarrow$ longest prefix of w already in the TRIE. That is where $\delta(q_0, \alpha) \neq \emptyset$,
   (c) $\beta \leftarrow$ remaining suffix of w where $w = \alpha\beta$,
   (d) **if** $\beta \neq \epsilon$
   add the states representing the suffix β to the state $\delta(q_0, \alpha)$,
   (e) add last state in the inserted word to the final state set so that $F = F \cup \delta(q_0, w)$.

In step $3(d)$ of the algorithm above we add the suffix $\beta = \beta_0 \ldots \beta_{m-1}$ to the state $\delta(q_0, \alpha)$ as follows:

- Add m states,
$$\{q_0^\beta, \ldots, q_{m-1}^\beta\},$$
to Q where $m = |\beta|$ (that is, the number of characters in β), and
- add the following m transitions to δ with $q' = \delta(q_0, \alpha)$:

$$\begin{array}{lcl} \delta(q', \beta_0) & = & q_0^\beta, \\ \delta(q_{i-1}^\beta, \beta_i) & = & q_i^\beta \quad , \text{ for } i = 1, \ldots, m-1. \end{array}$$

[4]For further information on TRIEs or suffix trees see [13].

Example 4.9 (TRIE)

As an example we construct the TRIE $(Q, \{0,1,2,3\}, \delta, q_0, F)$ *from the set of words:*

$$0,\ 001,\ 020,\ 101,\ 112,\ 212,\ 301,\ 312.$$

We create the TRIE as follows (also shown in figure 4.1):

1. *Start by creating the state* q_0 *and adding the word* 0 *to state* q_0.
2. *Next word:* 001. *As* 0 *is a prefix of* 001, *the suffix* 01 *is added to the state* $\delta(q_0, 0)$.
3. *Next word:* 020. *The longest common prefix is* 0, *and hence the suffix* 20 *is added to the state* $\delta(q_0, 0)$.
4. *Next word:* 101. 101 *shares no common prefix with the TRIE, so that* 101 *is added to the state* q_0.
5. *Next word:* 112. *The longest common prefix is* 1, *and hence the suffix* 12 *is added to the state* $\delta(q_0, 1)$.
6. *Similarly for the remaining words.*

The algorithm for the construction of a TRIE does not attempt to construct a minimal TRIE. As such the resultant TRIE or DFA contains reducible states. One such case of these reducible states is the final states which represent the end of words with no further transitions from them. These states are all equivalent to each other as their successors consist only of the word ϵ. These states may all be combined into one state reducing the number of states in the dictionary. Every state constructed during the TRIE construction is used in a word accepted by the FA. As such, no initial-useless or final-useless states are generated, thus only equivalent states remain.

The first algorithm minimizes the dictionary after the full TRIE has been constructed as above, this algorithm falls under the state comparison minimization category. This first algorithm, which we discuss in the next section, is used as a basis for the two language construction minimization techniques.

TRIE Minimization

As mentioned above the first algorithm minimizes a constructed TRIE. A TRIE is minimized by merging the equivalent states. Remember that a state is equivalent to another state if their successors are equal. In a TRIE a state's successor is equal to the suffixes from the state. This means that any states which share all of their suffixes are equivalent.

For example, in figure 4.1 of example 4.9 we note that the words 2[12] and 1[12] have the same set of suffixes, namely 12. Combining the states

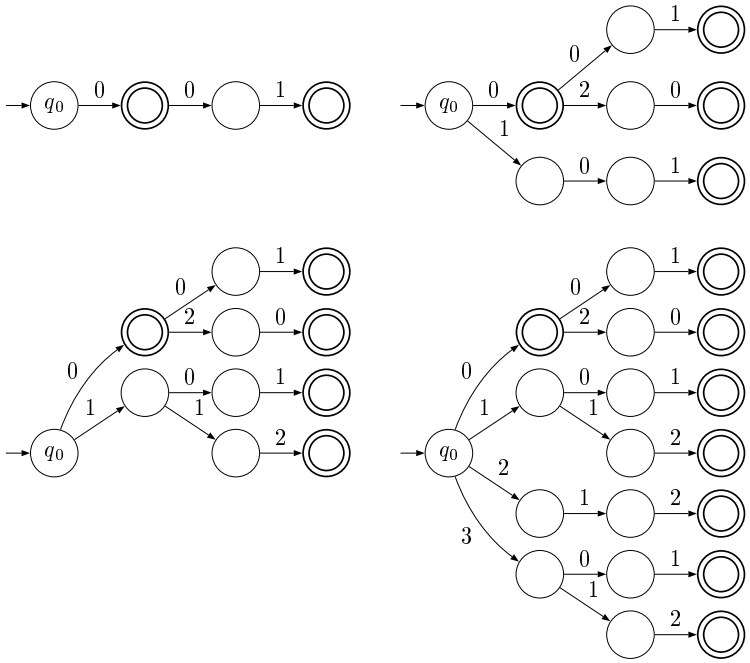

Figure 4.1: Example 4.9. Top Left: Step 2. Top Right: Step 4. Bottom Left: Step 5. Bottom Right: Step 6.

with predecessors 11 and 21 as well as 112 and 212 we obtain the DFA as shown in figure 4.2.

The algorithm finds these states with equal suffixes by moving through the TRIE layer by layer from the leaves to the root. A layer consists of states where the lengths of longest suffixes are equal. The first layer consists of only leaves, or states with no transitions from them. As these states have only one suffix each, namely ϵ, they are equivalent and can be merged into a single state. After these states are combined the next layer, states where the length of longest suffix is one greater than the previous layer, is processed.

Note that the length of the longest suffix from a child of a state is always smaller than that of the parent state. This implies that the children of the states of a layer being processed have all been processed.

Two states q_i, q_j are combined in a level if either both states are final or not, $q_i, q_j \in F$ or $q_i, q_j \notin F$, and they share all of their transition, $\delta(q_i, \sigma) = \delta(q_j, \sigma)$ for every $\sigma \in \Sigma$. Where this is the case we have that if $x \in sc(q_i)$ with $x \in \Sigma^*$ and $x = \sigma x'$:

$$\begin{aligned} \delta(q_i, x) &= \delta(\delta(q_i, \sigma), x') \\ &= \delta(\delta(q_j, \sigma), x') \\ &= \delta(q_j, x) \qquad . \end{aligned}$$

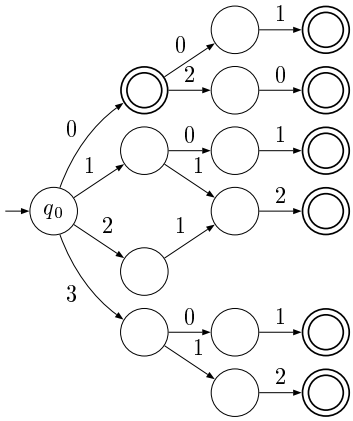

Figure 4.2: Combined states of dictionary from example 4.9.

Similarly, if $x \in sc(q_j)$ we have that $\delta(q_j, x) = \delta(q_i, x)$ so that $sc(q_i) = sc(q_j)$. As the states are equivalent they can be combined and the TRIE reduced. This comparison and merging of states is repeated for every layer.

The TRIE resulting from this algorithm is minimal as shown in [4]. Before continuing discussing TRIE minimization we provide an example of the first algorithm.

Example 4.10 (TRIE Minimization)
The TRIE from example 4.9 can be minimized using the first algorithm as discussed above.

The algorithm functions on the TRIE as follows (also shown in figure 4.3):

1. *Combine all the leaves.*

2. *Process the first level, that is where the length of the state's longest suffix is of length* 1. *The first layer consists of the states with prefixes* 00, 02, 10, 11, 21, 30 *and* 31. *The states with the prefixes* 00, 10 *and* 30 *all have a single suffix, namely* 1, *which are equal and can be combined. Similarly, the states with the prefixes* 11 *and* 31 *all have a single suffix, namely* 2, *which are equal and can be combined.*

3. *Process the second level, that is where the length of the state's longest suffix is of length* 2. *The second layer consists of the states with prefixes* 0, 1, 2 *and* 3. *The states with the prefixes* 1, *and* 3 *share their children, the states represented by the prefixes* 11 *or* 31 *and* 10 *or* 30. *As shown above this means that the states* 1 *and* 3 *share all suffixes,* 01 *and* 12, *and can be combined.*

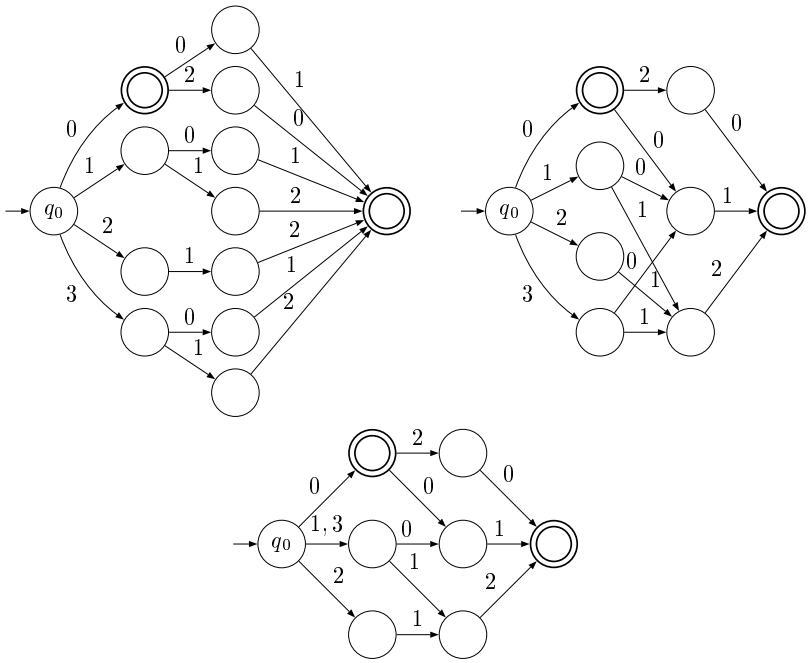

Figure 4.3: Example 4.10. Left: Step 1. Right: Step 2. Bottom: Step 4

4. *Process the third level, that is where the length of the state's longest suffix is of length* 3*. This layer consists only of the root, that is the state with the prefix* ϵ*.*

The resultant TRIE as shown in figure 4.3 is minimal.

The TRIE minimization algorithm can be run with worst case computational complexity $\mathcal{O}(n^2)$ where n is the number of states in the TRIE. This worst case computational complexity is a result of processing only a state's children and categorizing the states as the children are processed. By categorizing the states, they need not be explicitly compared to each other within a layer but instead can be automatically combined.

Although the algorithm can be run efficiently, the original TRIE must be generated and stored before use. It also implies that all the words must be inserted into the TRIE before minimization. If words were to be added after the minimized TRIE is generated, a new TRIE would need to be constructed and the minimization algorithm applied to the new TRIE. Two extensions to this algorithm presented by Daciuk, Watson and Watson [4] offer solutions to these drawbacks.

The first extension constructs the minimal TRIE from a sorted list of words. The TRIE is constructed by inserting each word separately and minimizing the states not shared between the new and previous word. The

construction, given the TRIE immediately after processing the word y and given the next word in alphabetical order x, is as follows:

> Insert the word x into the TRIE as done for TRIE construction. During this insertion only the states representing the suffix of x corresponding to the longest common prefix of x and y need to be added to the TRIE. As the words are sorted alphabetically we know that no word will be inserted with a longer common prefix with y than x. This implies that the states representing the remaining suffix of y will never be modified again and can be minimized using the original TRIE minimization algorithm.
>
> If they could be modified then there exists a word z after y in the list where z shares a larger prefix with x than y which as the list is sorted is a contradiction.

The TRIE representing the list of words up to and including the previously inserted word is thus minimized after inserting a new word. This minimization can be run in linear time as only a single suffix, that of the previously inserted word, needs be processed from the leaves back up the TRIE. This solution solves the space complexity of the original TRIE minimization algorithm and runs efficiently. However, sorting the words before insertion is not always practical. Examples include cases where large amounts of unsorted data are being streamed from an input device (such as DNA sequencing). It also does not allow words to be inserted at a later stage without reconstructing the entire TRIE from a new list.

The second extension makes provision for unsorted data. This allows unsorted lists to be used to generate the TRIE as well as allowing for inserting new words into the TRIE at a later stage. However, this results in a higher worst case computational complexity.

In the case of sorted data only the states representing the suffix of the previously inserted word needed to be processed for equivalence. However, this is not the case with unsorted data. Instead on the insertion of a new word, the longest prefix of the word in the TRIE is determined. The states representing the suffixes of this state are duplicated in the TRIE so that a new suffix can be added to the TRIE without affecting other words with the same suffixes but different prefixes. After the new suffix has been added the split states are checked for equivalence resulting in the minimal TRIE representing the inserted words.

This concludes our discussion of the minimization of the specialized form of DFAs known as dictionaries. The algorithms we discussed for dictionary minimization provided both efficient space and worst case computational complexity. However, as the dictionary is a simplified form of the DFA the process of detection and removal of reducible states was simplified to only

looking for equivalent state. This could be done efficiently by searching backwards from the generated common final state or leaf.

It is important to note that a DFA is a simplified form of a DFA due to the property that they contain no infinite length words. It follows that dictionary minimization algorithms which make use of this property will not be able to be used for the minimization of NFAs. Similarly, the DFA minimization algorithms we will discuss make use of properties of DFAs which are not shared by NFAs. With this in mind we begin our discussion of general DFA minimization algorithms. The first DFA minimization algorithm we discuss is by Hopcroft [7] which makes use of a simplified equivalence test.

4.2.2 Hopcroft's Algorithm

In this section we discuss two DFA minimization algorithms both proposed by Hopcroft [7]. The first algorithm divided the given DFA into disjoint sets used for detecting equivalent states. The worst case computational complexity of this algorithm is $\mathcal{O}(n^2)$ where n is the number of states in the original DFA. The second algorithm involves a modification on how these sets are compared and generated. This modification results in an improvement of the worst case computational complexity to $\mathcal{O}(n \log(n))$.

We note from the equivalence test in corollary 4.3 (page 31) that the worst case computational complexity of comparing the successors of states can be as complex as computing the language of the DFA. Such brute force techniques would be impractical in practice.

Hopcroft's algorithm divides the states of the DFA into equivalence sets. Each of these sets represents states which are possibly equivalent. States are tested for equivalence to only the other states in the set and removed if inequivalence is detected. Instead of directly comparing the successors of states, which would be computationally expensive, a simplified equivalence test is used which makes use of only the states' children.

This simplified test is based on the following lemma[5]:

Lemma 4.11 (Equivalence Test)
In a DFA $M = (Q, \Sigma, \delta, Q_0, F)$, $q_i \equiv q_j$ *where* $q_i, q_j \in Q$ *if and only if*

$$\delta(q_i, \sigma) \equiv \delta(q_j, \sigma)$$

for all $\sigma \in \Sigma$.

Therefore, a state in a DFA is equivalent to another state if and only if the children of the states are equivalent.

We highlight the equivalence test with the following example:

[5]See [3] for a proof of lemma 4.11.

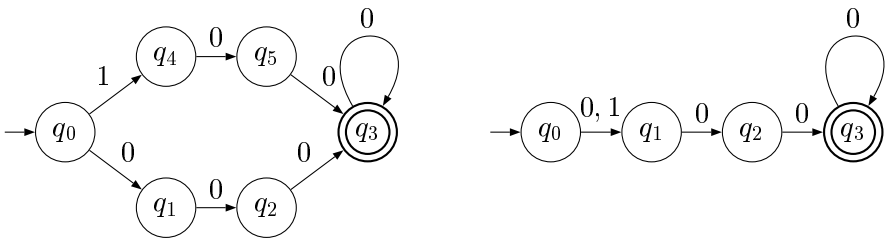

Figure 4.4: Example 4.12. Left: M. Right: M'

Example 4.12 (Equivalence Test)
Given the DFA $M = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{0, 1\}, \delta, \{q_0\}, \{q_3\})$ *with* δ *given by (also given in figure 4.4):*

δ	0	1
q_0	q_1	q_4
q_1	q_2	$\emptyset$
q_2	q_3	$\emptyset$
q_3	q_3	$\emptyset$
q_4	q_5	$\emptyset$
q_5	q_3	$\emptyset$

.

We observe that

$$\delta(q_2, 0) = q_3 = \delta(q_5, 0), \text{ and } \delta(q_2, 1) = \emptyset = \delta(q_5, 1).$$

As $q_3 \equiv q_3$ *from lemma 4.11 we have that* $q_2 \equiv q_5$.
Similarly

$$\delta(q_1, 0) = q_2 \equiv q_5 = \delta(q_4, 0), \text{ and } \delta(q_1, 1) = \emptyset = \delta(q_4, 1)$$

so that $q_1 \equiv q_4$.

Merging the equivalent states q_2, q_5 *and* q_1, q_4 *we construct the minimized DFA* $M' = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta', \{q_0\}, \{q_3\})$ *with* δ' *given by (also given in figure 4.4):*

δ'	0	1
q_0	q_1	q_4
q_1	q_2	$\emptyset$
q_2	q_3	$\emptyset$
q_3	q_3	$\emptyset$

.

With lemma 4.11 in mind we can discuss the original DFA minimization algorithm by Hopcroft. Note that the original proof for the algorithms are defined for output DFAs and are not recreated in this article[6].

[6]The published minimization algorithm by Hopcroft was defined for output DFAs, which are an extension of FAs. The definition and conversion of output FAs to standard FAs are given in A.9 and A.10, page 170.

Hopcroft's Algorithm

The algorithm function by first dividing the DFA into an initial pair of disjoint[7] equivalence sets. After this initial division of states the algorithm subdivides the equivalence sets until each set consists only of equivalent states. After the sets can no longer be subdivided the minimized DFA is constructed by combining the states within the equivalence sets.

Given a DFA M with $M = (Q, \Sigma, \delta, Q_0, F)$ the initial pair of disjoint equivalence sets are the sets

$$A_1 = F,\ A_2 = Q - F.$$

An equivalence set A_i is divided if the following condition is met:

- For any alphabet symbol $\sigma \in \Sigma$, and any two states $q_j, q_k \in A_i$, the destination states on the transitions of the alphabet symbol do not belong to the same equivalence set. So that an equivalence set is divided if for any two states in the equivalence set $q_j, q_k \in A_i$ we have that $\delta(q_j, \sigma) \in A_x$ and $\delta(q_k, \sigma) \in A_y$ with $x \neq y$.

When the above condition is met the equivalence set A_i is divided into two equivalence sets $A_{i'}$ and $A_{\cap i}$. $A_{i'}$ consists of all the states $q \in A_i$ where $\delta(q, \sigma) \in A_x$. Similarly $A_{\cap i}$ consists of all the states $q \in A_i$ where $\delta(q, \sigma) \notin A_x$.

Once no more divisions can be made in any equivalence set we can construct the minimal DFA is constructed. In [7] lemma 4.11 is used to show that the remaining states in the equivalence sets are equivalent with each other. It follows that the minimal DFA can then be constructed by merging each equivalence set into a single state.

Note that in the algorithm i represents a counter, A_i represents the i'th equivalence set and m represents the number of equivalence classes.

Algorithm 4.13 (Hopcroft's Algorithm)

Given a DFA $M = (Q, \Sigma, \delta, Q_0, F)$ the algorithm outputs all the equivalence sets A_i of M.

(1) Initialization. $A_1 \leftarrow F$, $A_2 \leftarrow Q - F$, $m \leftarrow 2$, $i \leftarrow 0$.

(2) while $(i < m)$ *do*

(a) $i \leftarrow i + 1$.

(b) Assign q to any state in A_i.

(c) if ($\exists$ a state $q' \in A_i$ so that $\delta(q, \sigma) \in A_k$ and $\delta(q', \sigma) \notin A_k$) then

$$A_{m+1} \leftarrow \{q \mid q \in A_i,\ \delta(q_i, \sigma) \notin A_k\},$$

[7] Disjoint sets share no elements.

$A_i \leftarrow A_i - A_{m+1}$,
$m \leftarrow m+1$, $i \leftarrow 0$.

(3) Return $A_1, \ldots, A_m$.

Note 4 *In step 2b of algorithm 4.13 any state from* A_i *is selected. The state is then compared to every other state in the equivalence set. The results of the logical test in step 2c will be identical irrespective of the state chosen in step 2b. So that, either all the states' children share equivalence sets or at least one state does not. The states whose children do not share equivalence sets will be detected through the test in step 3c. Thus only one state needs to be selected for each equivalence set per iteration of step 2.*

The algorithm has a worst case computational complexity of $O(n^2)$ as shown in [7]. This follows from the case where no states in the DFA are equivalent. In this case there will be $|Q| = n$ equivalence sets, so that step 3 may be repeated at most $\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$ times.

Before discussing the modified algorithm we provide the following example of Hopcroft's algorithm.

Example 4.14 (Original Hopcroft Algorithm)
Let M *be the DFA* $M = (\{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_1\})$ *with* δ *described by (also shown in figure 4.5)*

δ	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_3	q_2
q_3	q_3	q_1

.

We start by dividing Q *into two disjoint equivalence sets based on the final states:*

$$\{q_0,\ q_2,\ q_3\},\ \{q_1\}.$$

It follows that q_1 *is not equivalent to any other state as it is the only element in an equivalent state set. However, the states* q_0, q_2 *and* q_3 *may all be equivalent to each other.*

Examining the equivalence set $\{q_0, q_2, q_3\}$ *on input symbol* 1 *we find that* $\delta(q_2, 1) \in \{q_0, q_2, q_3\}$, *while* $\delta(q_0, 1) = \delta(q_3, 1) \in \{q_1\}$. *From the algorithm we must therefore divide the set* $\{q_0, q_2, q_3\}$ *to obtain the following sets:*

$$\{q_0, q_3\},\ \{q_2\},\ \{q_1\}.$$

At this point no further divisions can be made.

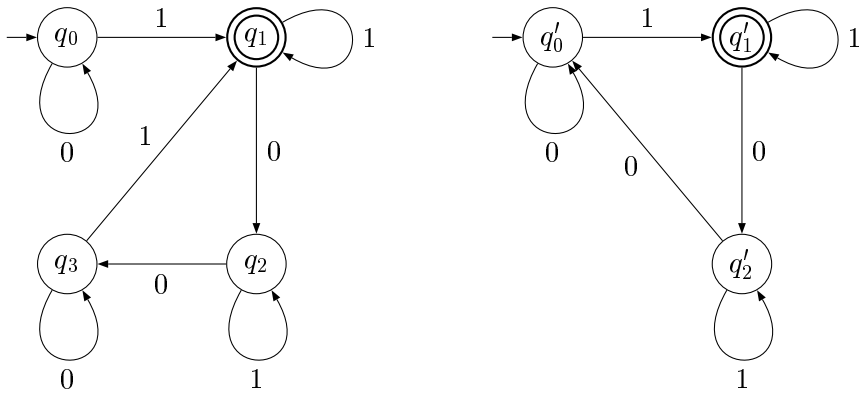

Figure 4.5: Example 4.14: Left M, Right M'

The resultant minimal DFA is $M' = (\{q'_0, q'_1, q'_2\}, \Sigma, \delta', q'_0, \{q'_1\})$ *where* δ' *can be described as (also shown in figure 4.5)*

δ'	0	1
q'_0	q'_0	q'_1
q'_1	q'_2	q'_1
q'_2	q'_0	q'_2

.

The state q'_0 *in* M' *represents the equivalence set* $\{q_0,\ q_3\}$ *from the algorithm.*

This concludes our discussion of Hopcroft's original minimization algorithm. We now discuss the modified algorithm.

Hopcroft's Modified Algorithm

As discussed in the previous section, the original Hopcroft algorithm has a worst case computational complexity of $\mathcal{O}(n^2)$ for an n-state DFA. In fact as shown in [19] most DFA minimization algorithms based on the detection and removal of equivalent states have a worst case computational complexity of $\mathcal{O}(n^2)$. By modifying the way in which the algorithm divides the equivalence sets a time complexity of $O(n \log n)$ can be obtained [7] .

In the original algorithm each state in the equivalence set has it children compared with every other state's children in the equivalence set. The modified algorithm compares the parents of the states in the equivalence sets rather then their children. An equivalence set, A_i, is divided if the set of parents of an equivalence set A_j which are elements of A_i form a true subset of A_i.

For example when examining an equivalence set A_i, Given the parents of the states of the equivalence set A_j as the set of states A'. A_i is divided if $A' \bigcup A_i \subset A_i$ and $A' \neq \emptyset$.

To simplify the process of finding the parent of a state on a given alphabet symbol the algorithm first generates the parent state table. The parent state table maps states, and alphabet symbols to the parent of the state on the alphabet symbol. When examining an equivalence set the parent state table can be referenced rather than searching for the parents of the states. Note that a state may have multiple parents on a given alphabet symbol in a DFA.

To highlight these changes we provide the following example before discussing the modified algorithm.

Example 4.15 (Hopcroft's Modified Algorithm)
Let M be the DFA from example 4.14 (page 45).
The parent state table can be constructed as follows:

	0	1
q_0	q_0	$\emptyset$
q_1	$\emptyset$	$\{q_0, q_1, q_3\}$
q_2	q_1	q_2
q_3	$\{q_2, q_3\}$	$\emptyset$

.

The algorithm starts by dividing Q into the two disjoint equivalence sets:

$$\{q_0, q_2, q_3\},\ \{q_1\}.$$

On input symbol 1 the equivalence set $\{q_1\}$ has as parents the state set $pa(q_1) = \{q_0, q_1, q_3\}$. As $\{q_0, q_1, q_3\} \cap \{q_0, q_2, q_3\} = \{q_0, q_3\} \subset \{q_0, q_2, q_3\}$, the equivalence set $\{q_0, q_2, q_3\}$ is divided to obtain the following equivalence set:

$$\{q_0, q_3\},\ \{q_2\},\ \{q_1\}.$$

No further divisions can be made. The minimal DFA can then be constructed by merging states within equivalence classes as done for example 4.14.

Note that the detection of the divisions is not unique. In the example the division can be obtained by examining the equivalence set $\{q_0, q_2, q_3\}$ on input 0.

In the original algorithm we continually divide the set we were examining. However, in the modified algorithm we divide only equivalence sets containing subsets of the parents to the examined equivalence set on the given alphabet symbol. It follows that once an equivalence set has been examined on a specific input symbol it never needs to be examined on that input symbol again, unless the equivalence set is divided.

Examining an equivalence set requires the examination of all equivalence sets containing parents of the equivalence set, so that at most n iterations are needed.

Furthermore, since the partitioning of an equivalence set is proportional to the number of transitions to the states of the equivalence set we can

always select the parent subset with fewer transitions. The total resultant worst case computational complexity of the algorithm is therefore $O(n \log n)$ as shown in [7].

Hopcroft's modified algorithm uses the following structures for the computation of the equivalence classes:

- A_i represents the i'th equivalence set, and $\mathcal{Z}$ is the set of integers,
- $k-1$ is the number of equivalence sets,
- $\overleftarrow{\delta}$ is known as the inverse transition function and is defined for the states $q_i, q \in Q$ and $\sigma \in \Sigma$ as:

$$q_i \in \overleftarrow{\delta}(q, \sigma) \;\Leftrightarrow\; q \in \delta(q_i, \sigma),$$

- $s(\sigma, i)$ is a function $s : \Sigma, \mathcal{Z} \rightarrow 2^Q$ which maps the set of states in the equivalence set A_i with parents on the alphabet symbol α, and
- $L(\sigma)$ is a function $L : \Sigma \rightarrow \mathcal{Z}$ which maps the alphabet symbols to the index of the equivalence set with the fewest number of parents on the alphabet symbol σ.

Note that in the algorithm when an equivalence set is divided, the structures s and L have to be updated. It is important to note that as only one equivalence set needs to be added to L the one with fewer parent states, determined by s can be chosen. However, in the case where the divided equivalence set was in L, both sets are added to L.

With these structures we can formally define the method as follows:

Algorithm 4.16 (Hopcroft's Modified Algorithm)

Given a DFA $M = (Q, \Sigma, \delta, Q_0, F)$, the algorithm outputs all the equivalence sets, A_i, of M as follows:

(1) Initialization. $A_1 \leftarrow F$, $A_2 \leftarrow Q - F$, $k \leftarrow 3$.

(2) Construct $\overleftarrow{\delta}$.

(3) Construct $s(\sigma, i)$ *for* $i \leftarrow 1 \ldots 2$ *and every* $\sigma \in \Sigma$.

(4) Construct $L(\sigma)$ *for all* $\sigma \in \Sigma$.

(5) While ($\exists \sigma \in \Sigma$ *so that* $L(\sigma) \neq 0$)

(a) Select any $i \in L(\sigma)$.

(b) $L(\sigma) \leftarrow L(\sigma)/\{i\}$.

(c) For $j \leftarrow 1 \ldots k-1$ *do*

If ($\exists q \in A_j$ *where* $\delta(q, \sigma) \in s(\sigma, i)$) *then*

- $A_k \leftarrow \{q \mid \delta(q,\sigma) \notin s(\sigma,i)\}$,
- $A_j \leftarrow A_j / A_k$,
- $s(\sigma,j) \leftarrow \{q' | q' \in A_j,\ where\ \overleftarrow{\delta}(q',\sigma) \neq \emptyset\}$,
- $s(\sigma,k) \leftarrow \{q' | q' \in A_k,\ where\ \overleftarrow{\delta}(q',\sigma) \neq \emptyset\}$,
- $\forall \sigma \in \Sigma$
  if $j \notin L(\sigma)$ *and* $0 < |s(\sigma,j)| \leq |s(\sigma,k)|$ *then*
  $L(\sigma) \leftarrow L(\sigma) \cup \{j\}$
  else
  $L(\sigma) \leftarrow L(\sigma) \cup \{k\}$.
- $k \leftarrow k+1$.

(6) Return $A_1, \ldots, A_{k-1}$.

Although the algorithm appears to differ greatly from its original, only the detection of equivalence sets which need to be divided is different. As shown in [19], Hopcroft's modified algorithm achieves the fastest minimization worst case computational complexity for all DFA minimization algorithms and represents the most advanced of such algorithms. For more information into similar algorithms the reader may refer to Watson [19].

This concludes our discussion of Hopcroft's DFA minimization algorithms. Although the algorithm is the most efficient among DFA minimization algorithm [19] it uses a simplified equivalence test which is not useable for NFAs. The final DFA minimization algorithm we discuss by Brzozowski [2] makes use of transformation construction. Although not directly useable as an NFA minimization algorithm, Brzozowski's algorithm follows a unique approach to minimization which is used in an NFA minimization algorithm we discuss later.

4.2.3 Brzozowski's Algorithm

The previous DFA minimization algorithms both detected and removed reducible states either during construction or given a DFA. Brzozowski's algorithm [2] is unique in the sense that it does not detect these reducible states, but instead removes them from the DFA by applying various transformations to the DFA. It does this by making use of on-the-fly subset construction from algorithm 2.11 (page 13).

As we discussed at the beginning of this chapter, DFAs do not have redundant states consisting only of initial-useless, final-useless and equivalent states. Thus transforming a FA into an equivalent DFA through subset construction removes all redundant states, without identifying them. On-the-fly subset construction goes one step further by not generating any initial-useless states, leaving only final-useless and equivalent states.

Brzozowski's algorithm makes use of one other transformation, namely the dual FA [11], which recognizes the inverse language[8] of a FA.

The dual of a FA is defined as follows:

Definition 4.17 (Dual) *The **dual** of a FA* $M = (Q, \Sigma, \delta, Q_0, F)$ *is a FA* $\overleftarrow{M} = (Q, \Sigma, \overleftarrow{\delta}, F, Q_0)$ *where*

$$q_i \in \overleftarrow{\delta}\ (q_j, \sigma) \Leftrightarrow q_j \in \delta(q_i, \sigma)$$

with $q_i, q_j \in Q$ *and* $\sigma \in \Sigma$.

That is, in the dual of a FA all transitions are reversed, so that a transition to a state becomes a transition from the state. Similarly the initial state set is exchanged with the final state set.

Before discussing properties of the dual FA we note that the dual of a DFA with more than one final state has multiple initial states. Thus a dual of a DFA may be an NFA.

The proof for the following lemma which deals with the predecessors and successors of a dual is given in [11].

Lemma 4.18 (Dual Languages)
Given a FA $M = (Q, \Sigma, \delta, Q_0, F)$ *with dual* $\overleftarrow{M} = (Q, \Sigma, \overleftarrow{\delta}, F, Q_0)$, *it holds that*

$$\begin{aligned} pr_M(q) &= \overleftarrow{sc}_{\overleftarrow{M}}(q) \\ sc_M(q) &= \overleftarrow{pr}_{\overleftarrow{M}}(q) \end{aligned}$$

with $q \in Q$.

As shown in [11] it follows from the lemma and definition 2.5 (page 9) that by combining the successors of the final states in the dual of a FA, $\mathcal{L}(M) = \overleftarrow{\mathcal{L}}(\overleftarrow{M})$, and that $\overleftarrow{\overleftarrow{M}} \equiv M$ or the dual of the dual of a FA is a FA.

It follows from the lemma and the definitions of reducible states in definition 4.4 (page 31), that the problem of removing equivalent and final-useless states from a FA M is equivalent to removing redundant and initial-useless states in $\overleftarrow{M}$. We note that DFAs constructed using on-the-fly subset construction remove redundant and initial-useless states. This implies that the DFA resulting from on-the-fly subset construction of the dual of the DFA is minimal. Similarly the DFA resulting from on-the-fly subset construction of the dual of the minimal dual DFA.

It is important to note that Brzozowski's algorithm functions on all FAs, not just DFAs.

With this in mind we examine Brzozowski's algorithm:

[8] See definition A.11, page 171.

Algorithm 4.19 (Brzozowski)

Given a FA $M = (Q, \Sigma, \delta, Q_0, F)$*, the algorithm returns the equivalent minimal DFA* $M' = (Q', \Sigma, \delta', q_0', F')$ *as follows:*

1. *Construct NFA* $A \leftarrow \overleftarrow{M}$ *as per definition 4.17 (page 50).*
2. *Construct DFA* $A' \leftarrow \mathrm{DFA}(A)$.
3. *Construct NFA* $B \leftarrow \overleftarrow{A'}$.
4. *Construct DFA* $M' \leftarrow \mathrm{DFA}(B)$.
5. *Return* M'.

The algorithm functions by first generating the DFA of the dual of the FA. The first two steps remove the equivalent and final-useless states from M. Similarly, steps 3 and 4 remove any redundant and initial-useless states.

But is the resultant DFA minimal?
We show this by stepping through the algorithm, examining the languages of each FA in turn:

Firstly we note that

$$\mathcal{L}(A') \;=\; \mathcal{L}(A) \;=\; \overleftarrow{\mathcal{L}}(M),$$

since, as A is the dual of M it accepts the dual language of M. Furthermore, A' is equivalent to A and so accepts the same language.

Then,

$$\mathcal{L}(M') \;=\; \mathcal{L}(B) \;=\; \overleftarrow{\mathcal{L}}(A'),$$

and as B is the dual of A it accepts the dual language of M. Furthermore, M' is equivalent to B and so accepts the same language.

Combining these properties we have that

$$\mathcal{L}(M') \;=\; \overleftarrow{\overleftarrow{\mathcal{L}}}(M) \;=\; \mathcal{L}(M).$$

So that, M and M' are equivalent. Through on-the-fly subset construction we have that M' has no reducible states so that the resultant DFA is the minimal DFA equivalent to M.

Although elegant, Brzozowski's algorithm makes use of subset construction which has a worst case space and worst case computational complexity of $\mathcal{O}(n^2)$ where n is the number of states in the DFA. As the dual can be computed in linear time depending on how the DFA is stored, Brzozowski's algorithm has a worst case time and worst case computational complexity of $\mathcal{O}(n^2)$.

However, Brzozowski's algorithm is still of interest to us due to its use of the dual and the fact that DFAs do not have redundant states. Kameda and

Weiner's algorithm [11] for NFA minimization[9] makes use of Brzozowski's algorithm, paying special attention to the dual.

This concludes our discussion of Brzozowski's algorithms. Before discussing NFA minimization algorithms we discuss various properties of the minimal DFAs.

Before continuing onto NFA minimization we discuss some properties of minimal DFAs which are of interest to us.

4.2.4 Minimal DFAs

In the previous section we discussed three methods of constructing minimal DFAs. These methods form a basis of our discussion of NFA minimization algorithms and for $\oplus$-FA minimization. The algorithms we discussed were

- Dictionary minimization, based on language reconstruction;
- Hopcroft's algorithm, based on state comparison; and
- Brzozowski's algorithm, based on transformation construction.

Before continuing onto the minimization of NFAs, it will be of interest to examine certain properties of minimal DFAs.

The first property we discuss is that a minimal DFA is unique. This will be of use to us in determining the limits of the size of the minimal NFA.

Theorem 4.20 (DFA Uniqueness)
A minimal DFA is unique up to isomorphism[10].

Proof See [11]. □

Theorem 4.20 implies that we can detect whether a DFA is minimal by comparing it isomorphically to the resultant DFA of one of the above DFA minimization algorithms. Furthermore, each language will have a unique minimal DFA which accepts it. By comparing the minimal DFAs equivalent to given FAs we can determine whether the original FAs are equivalent without directly comparing their languages.

We use the above theorem to determine the size of a minimal NFA which accepts a given language.

Theorem 4.21 (NFA Range Bound)
Given the minimal DFA M' with n states accepting the language $L(M')$, a minimal NFA, $M = (Q, \Sigma, \delta, Q_0, F)$, accepting the same language has

$$\log_2(n) \leq |Q| \leq n$$

states.

[9] As discussed in section 4.3.1 (page 56).
[10] See definition A.12, page 171.

Proof Proof by contradiction.

Let $M' = (P, \Sigma, \delta', p_0, F')$ to be a minimal DFA with $|P| = n$.

Assume $M = (Q, \Sigma, \delta, Q_0, F)$ is a minimal NFA which accepts the same language as M'. If $|Q| > n$ then we have that $|Q| > |P|$. But as M' is a DFA, it is also an NFA so that an equivalent NFA exists with a smaller set of states, and hence M is not minimal. Thus by contradiction $|Q| \leq n$.

Next assume that $|Q| < \log_2(n)$. Then, by using the subset construction algorithm on M, we obtain $M'' = (P'', \Sigma, \delta'', p_0'', F'')$. As $P'' \subseteq 2^Q$ we have that

$$|P''| \leq 2^{|Q|} < 2^{\log_2(n)} = n.$$

But if $|P''| < n$, then M'' is a DFA with a smaller set of states than M', so that M' is not minimal. Thus by contradiction $|Q| \geq \log_2(n)$. □

Theorem 4.21 defines the bounds of the size of a minimal NFA given the equivalent minimal DFA. It follows that given a minimal DFA with n states, an equivalent NFA with $\lceil \log_2(n) \rceil$ is minimal.

It is of interest which minimal NFAs are exactly of size n and $\log_2(n)$ respective to the equivalent minimal DFA.

In example 2.8 on page 11 we have a minimal DFA equivalent to a minimal NFA of the same size. We will also provide some later examples where the minimal NFA has a state size of $\log_2(n)$ where n is the state size of the minimal DFA.

This concludes our discussion on the minimization of DFAs. In the next section we discuss NFA minimization and reduction algorithms.

4.3 NFA Minimization

In this chapter we have discussed general concepts of reduction and three DFA minimization algorithms. As discussed, DFA minimization algorithms may be elegant, as in the case of Brzozowski's minimization algorithm (Section 4.2.3), or run in polynomial time, as for Hopcroft's algorithm (Section 4.2.2). However, as indicated in these algorithms similar elegance and worst case computational complexity results are not possible for NFAs. This is due to the added complexity of the non-deterministic transition function. As shown by Jiang and Ravikumar [10] the problem of NFA minimization is NP hard.

A solution to this problem is the use of NFA reduction algorithms. These algorithms do not guarantee a minimal NFA but can be computationally inexpensive as shown in [9]. These reduction algorithms are also of use for DFAs as it reduces the number of states in the NFA for use with subset construction.

We discussed three DFA minimization algorithms, representing language reconstruction, state comparison or transformation reconstruction methods.

NFA minimization algorithms also exist for each of these methods, such as the NFA language reconstruction algorithm by Hagenah and Muscholl [6]. As we have covered the basics of FA minimization and are more interested in NFA minimization algorithms which can be used as a basis for $\oplus$-FA minimization we do not represent minimization algorithms for all three categories. Instead we discuss two algorithms. The first algorithm we discuss is a minimization algorithm by Kameda and Weiner [11] based on transformation reconstruction.

After discussing Kameda and Weiner's minimization algorithm we examine the reducible states of NFAs.

We the continue by discussing a set of reduction algorithms by Ilie and Yu [8], Champarnaurd and Coulon [3], and Ilie, Navarro and Yu [9]. Each reduction algorithm is in turn based on the previous reduction algorithm. These algorithms make use of simplified reducible state tests and can be run with a linear worst case computational complexity.

NFA minimization algorithms deal with the combination and removal of reducible states in a similar method for DFAs in definitions 4.1, and 4.4 (pages 29, 31). Where in DFAs a state may only be reducible in terms of another state, in the case of NFAs the active state set may consist of multiple final states and so states may be reducible in terms of set of states.

The definitions of reducible states are extended for use with NFAs as follows:

Definition 4.22 (Reducible States)
In an NFA $M = (Q, \Sigma, \delta, Q_0, F)$ with a state $q \in Q$:

- *q is **equivalent** to a set of states $Q' \subset Q$ with $q \notin Q'$ if*

$$sc(q) = sc(Q') = \bigcup_{q' \in Q'} sc(q'),$$

- *q is **redundant** with respect to a set of states $Q' \subset Q$ with $q \notin Q'$ if*

$$pr(q) = \bigcup_{q' \in Q'} pr(q'),$$

- *q is **contained** by a set of states $Q' \subset Q$ with $q \notin Q'$ if*

$$pr(q) \subseteq \bigcup_{q' \in Q'} pr(q') \text{ and } sc(q) \subseteq sc(Q').$$

An NFA $M = (Q, \Sigma, \delta, Q_0, F)$ with reducible states as defined in definition 4.22 can be reduced as follows:

- If a state $q \in Q$ is equivalent to a set of states $Q' \subset Q$ then the NFA can be reduced by removing the state q from the state set, initial state

set and final state set of the NFA. Furthermore, any transitions to the removed state q are replaced with transitions to the states of the set Q'.

Formally, if a state $q \in Q$ is equivalent to a set of states $Q' \subset Q$ then $\mathcal{L}(M) \equiv \mathcal{L}(M')$ with the NFA $M' = (Q/\{q\}, \Sigma, \delta', Q'_0, F')$ where:

- $\delta'(q_i, \sigma) = \begin{cases} \delta(q_i, \sigma)/\{q\} \cup Q' & \text{if } q \in \delta(q_i, \sigma), \\ \delta(q_i, \sigma) & \text{if } q \notin \delta(q_i, \sigma), \end{cases}$
- $Q'_0 = Q_0/\{q\} \cup Q'$ if $q \in Q_0$ otherwise $Q'_0 = Q_0$, and
- $F' = F/\{q\}$.

- If a state $q \in Q$ is redundant with respect to a set of states $Q' \subset Q$ then the NFA can be reduced by removing the state q from the state set and replacing the occurrence of q in the initial state set or final state set with the set Q'. Furthermore, the transitions from the removed state q are added to the transitions from all the states $q_i \in Q'$.

  Formally, if a state $q \in Q$ is redundant with respect to a set of states $Q' \subset Q$ then $\mathcal{L}(M) \equiv \mathcal{L}(M')$ with the NFA $M' = (Q/\{q\}, \Sigma, \delta', Q'_0, F')$ where:

  - $\delta'(q_i, \sigma) = \begin{cases} \delta''(q_i, \sigma) \cup \delta''(q, \sigma) & \text{if } q_i \in Q', \\ \delta''(q_i, \sigma) & \text{otherwise.} \end{cases}$
    where $\delta''(q_i, \sigma) = \delta(q_i, \sigma)/\{q\}$,
  - $Q'_0 = Q_0/\{q\}$, and
  - $F' = F/\{q\} \cup Q'$ if $q \in F$ otherwise $F' = F$.

- If a state $q \in Q$ is contained by a set of states $Q' \subset Q$ then the NFA can be reduced by removing the state q from the state set, initial state set and final state set of the NFA. Furthermore, all transitions from and to the removed state q are removed from the transition function.

  If a state $q \in Q$ is contained by a set of states $Q' \subset Q$ then $\mathcal{L}(M) \equiv \mathcal{L}(M')$ with the NFA $M' = (Q/\{q\}, \Sigma, \delta', Q'_0, F')$ where:

  - $\delta'(q_i, \sigma) = \delta(q_i, \sigma)/\{q\}$,
  - $Q'_0 = Q_0/\{q\}$, and
  - $F' = F/\{q\}$.

Where for DFAs in the worst case each state needed to be compared to every other state, for NFAs states may need to be compared to sets of states. This increase in the number of comparisons increases the worst case computational complexity of NFA minimization algorithms. NFA reduction

algorithms tend to ignore reducible states with regards to sets of states, resulting in a smaller worst case computational complexity.

It is important to note that the minimal NFA is not necessarily unique. This means that the order in which reducible states are removed may influence the resultant minimal NFA.

With these concepts in mind we start our discussion with the NFA minimization algorithm by Kameda and Weiner, before discussing the NFA reduction algorithms by Ilie, Navarro, Yu, Champarnaurd and Coulon.

4.3.1 Kameda and Weiner's algorithm

This algorithm was first presented in 1970 and as mentioned before is based upon the subset construction algorithm. Kameda and Weiner attempted to reconstruct a minimal NFA from the equivalent minimal DFA through the use of an inverse transformation to the subset construction algorithm.

In their paper [11] Kameda and Weiner provide a generalized algorithm to construct an equivalent NFA from a given DFA. From this point they provided an algorithm to find at least one mapping of states to construct a minimal NFA.

The algorithm attempts to construct a minimal NFA from the equivalent minimal DFA and as such a DFA minimization algorithm is needed. Kameda and Weiner opted for Brzozowski's minimization algorithm as they found the DFAs constructed during the minimization process are of use in the minimization algorithm.

We divide our discussion of Kameda and Weiner's minimization algorithm into two separate sections. We start by discussing the inverse subset construction defined as the **intersection rule**. After this we discuss the algorithm to find a mapping to generate a minimal NFA.

Intersection Rule

In this section we discuss the intersection rule[11] which can be described as the inverse operation to subset construction, that is, it creates an NFA from a given DFA.

We start by examining subset construction (Algorithm 2.11) given on page 13.

Given an NFA $M = (Q, \Sigma, \delta, Q_0, F)$ we have that $\text{DFA}(M) = M' = (P, \Sigma, \delta', p_0, F')$ with $p \in P$ iff $p \subseteq Q$. From this we have that every state in the DFA maps to a subset of states in the original NFA. The first step to reconstructing an NFA from a given DFA is to create a mapping of the states of the DFA to the states of the NFA states.

This mapping of states is defined as a subset assignment as follows:

[11] We describe this section in detail as we shall be using a modification of this algorithm for the minimization of symmetric difference finite automata in chapter 5.

Definition 4.23 (Subset Assignment)
*A **subset assignment** with $M' = (P, \Sigma, \delta', p_0, F')$ is defined as the pair $< Q, f >$ where Q is a set of states and $f : P \to 2^Q$ is the **subset assignment function**.*

Note 5 *As with the NFA bounds defined in 4.21, in a subset assignment the number of states in the NFA is range bound to*

$$\log_2 |P| \ \leq |Q| \ \leq |P|,$$

as each state in the DFA must map to a unique subset of Q.

For testing purposes of the intersection rule it is useful to use the original mappings of states from the NFA to the DFA. We define such a subset assignment function as the *natural subset assignment function.*

Therefore with $M = (Q, \Sigma, \delta, Q_0, F)$ and $M' = \text{DFA}(M) = (P, \Sigma, \delta', p_0, F')$ with $p \in P$ we have the natural subset assignment function as:

$$f(p) = \{q | q \in p\}.$$

Note that the natural subset assignment function can be defined on a DFA M' only if both the original NFA, M, and the relationship between the states in M and M' are known.

Next we use the subset assignment to determine the transition function of the constructed NFA. From the subset construction definition we know that with $p \in P$ and $\sigma \in \Sigma$ we have that:

$$\delta'(p, \sigma) \ = \bigcup_{q \in p} \delta(q, \sigma),$$

so that for a $q \in Q$, $q \in p$ means that

$$\delta(q, \sigma) \ \subseteq \ \delta'(p, \sigma).$$

Furthermore for every $p \in P$ where $q \in p$ we have that

$$\delta(q, \sigma) \bigcap_{q \in p} \delta'(p, \sigma) = \delta(q, \sigma),$$

so that

$$\delta(q, \sigma) \subseteq \bigcap_{q \in p} \delta'(p, \sigma).$$

By replacing the $\subseteq$ with an $=$ above we have that

$$\delta(q, \sigma) = \bigcap_{q \in p} \delta'(p, \sigma).$$

which can be used to generate an algorithm to construct the transition function of the NFA. Note that this modification no longer ensures the constructed NFA is equivalent to the DFA.

This means that when constructing an NFA from a given DFA and subset assignment, the transition function of the NFA on any state can be constructed from the intersection of the transition function of the DFA on states which contain the states of the NFA.

Lastly we need a way to determine the initial and final states of the constructed NFA. In the subset assignment definition we have that

$$p \in F' \text{ iff } \exists q \in p \text{ where } q \in F,$$

so that

$$q \in F \text{ iff } \forall p \text{ with } q \in p,\ p \in F'.$$

Using the concepts described above the intersection rule is defined as:

Definition 4.24 (Intersection Rule)
*Let $M' = (P, \Sigma, \delta', p_0, F')$ be a DFA, and let $< Q, f >$ be a subset assignment on M'. $I(Q, f, B)$ results in the NFA $M = (Q, \Sigma, \delta, Q_0, F)$ where I is the **Intersection Rule** and*

- $Q_0 = f(p_0)$,
- $q \in F \Leftrightarrow \forall q \in f(p),\ p \in F'$, *and*
- $q' \in \delta(q, \sigma) \Leftrightarrow \forall q \in f(p),\ q' \in f(\delta'(p, \sigma))$ *where* $\sigma \in \Sigma$.

Corollary 4.25
Let $M' = (P, \Sigma, \delta', p_0, F')$ be a DFA with $< Q, f >$ a subset assignment on M'. $I(Q, f, B)$ results in the NFA $M = (Q, \Sigma, \delta, Q_0, F)$ where

$$\delta(q, \sigma) = \bigcap_{p | q \in f(p)} f(\delta'(p, \sigma)), \quad \forall q \in Q,\ \forall \sigma \in \Sigma.$$

The following example uses the intersection rule to construct an NFA from a DFA.

Example 4.26
From example 2.2 (page 7) we have an NFA $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, \{q_1\}, \{q_3\})$ with δ given by

δ	0	1
q_0	$\{q_0\}$	$\{q_2\}$
q_1	$\{q_0, q_1\}$	$\{q_2\}$
q_2	$\{q_1\}$	$\{q_2\}$

,

from which we construct

$$M' = \text{DFA}(M) = (\{p_0, p_1, p_2, p_3\}, \{0, 1\}, \delta', p_0, \{p_1\})$$

and δ' *given by*

δ'	$\subseteq Q$	0	1
p_0	$\{q_0\}$	p_0	p_1
p_1	$\{q_2\}$	p_2	p_1
p_2	$\{q_1\}$	p_3	p_1
p_3	$\{q_0, q_1\}$	p_3	p_1 .

Let $< Q', f >$ *be the subset assignment with* $Q' = \{q'_0, q'_1, q'_2\}$ *where* f *is constructed based on the subset assignments and can be described as*

p	$f(p)$
p_0	q'_0
p_1	q'_1
p_2	q'_2
p_3	q'_0, q'_2 ,

so that $M'' = (Q', \{0,1\}, \delta'', Q'_0, F') = I(Q', f, M')$.

From definition 4.24 we have

$$\begin{aligned}
\delta''(q'_0, 0) &= f(\delta(p_0, 0)) \bigcap f(\delta(p_3, 0)) \\
&= \{q'_0\} \bigcap \{q'_0, q'_2\} \\
&= q'_0 \quad , \\
\delta''(q'_0, 1) &= f(\delta(p_0, 1)) \bigcap f(\delta(p_3, 1)) \\
&= \{q'_1\} \bigcap \{q'_1\} \\
&= q'_1 \quad , \\
\delta''(q'_1, 0) &= f(\delta(p_1, 0)) \\
&= q'_2 \quad , \\
\delta''(q'_1, 1) &= f(\delta(p_1, 1)) \\
&= q'_1 \quad , \\
\delta''(q'_2, 0) &= f(\delta(p_2, 0)) \bigcap f(\delta(p_3, 0)) \\
&= \{q'_0, q'_2\} \bigcap \{q'_0, q'_2\} \\
&= q'_0, q'_2 \quad , \textit{ and} \\
\delta''(q'_2, 1) &= f(\delta(p_2, 1)) \bigcap f(\delta(p_3, 1)) \\
&= \{q'_1\} \bigcap \{q'_1\} \\
&= q'_1 \quad .
\end{aligned}$$

Furthermore $Q'_0 = \{q'_0\}$ *and* $F' = \{q'_1\}$.

From figure 4.6 it can be seen that M'' *is isomorphic to* M, *so that* $M \equiv M''$.

In example 4.26 above the NFA resulting from the intersection rule on the subset assignment is isomorphic to the original NFA. This is not always the case, for example in the case where Q contains redundant states, that is if two states $q_i, q_j \in Q$ exist so that

$$q_i \in p \Leftrightarrow q_j \in p.$$

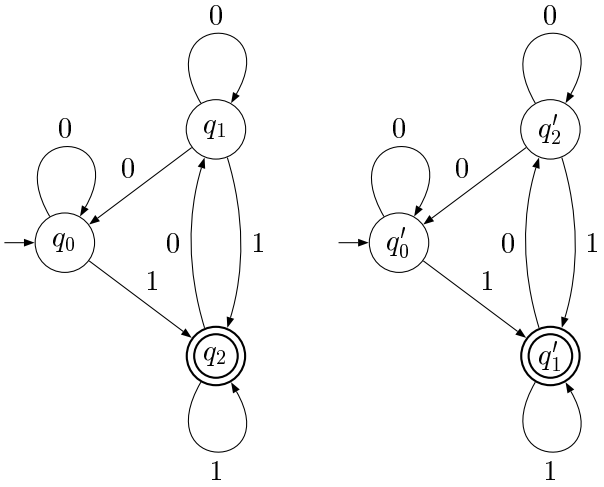

Figure 4.6: Example 4.26. Left: M, Right: M".

The intersection rule will not be able to differentiate between the two states. The algorithm will generate both states with the same set of incoming and outgoing transitions. Note that one of the redundant states can be removed by removing it from the state map.

As mentioned above the NFA constructed from the intersection rule is not always equivalent to the given DFA. For example constructing the NFA $M'' = I(Q', f, M')$, where M' is the DFA from example 4.26, with the subset assignment $< Q', f >$ where f is given by

p	$f(p)$
p_0	q'_0
p_1	q'_0, q'_1
p_2	q'_1, q'_2
p_3	q'_0, q'_2

.

We have that p_1 is the only final state in M'. A state q' in M'' is a final state if and only if for every state p in M', $q' \in f(p)$ if and only if p is a final state in M'. Using the example we have that the only final state in M' is p_2. As $f(p_2) = \{q'_0, q'_1\}$ but $q'_0 \in f(p_0)$ and $q'_1 \in f(p_2)$ so that neither q'_0 or q'_1 are final states. It follows that M'' has no final states so that $\mathcal{L}(M'') = \emptyset$.

From this short example we see that the intersection rule does not always result in an equivalent NFA. So how do we construct a subset assignment to guarantee the construction of an equivalent NFA? To answer this question we need the concepts which we will describe in the next section. Before discussing these concepts we first discuss why a brute force search would not be efficient in this situation.

Kameda and Weiner's algorithm makes use of the minimal DFA to reconstruct a minimal NFA. Remember that where the minimal DFA is of size n, the minimal NFA has a state size of between $\log_2(n)$ and n. The subset assignment $< Q, f >$ used in the algorithm has the property that Q is the smallest possible set of state which results in an equivalent NFA.

We can determine the number of possible subset assignments using the constraints on the size of a minimal NFA as follows: When constructing an NFA with m states from a DFA with n states there are ${}^{2^m}P_n$[12] possible subset assignments. This means that there are

$$\sum_{i=\lceil \log_2(n) \rceil}^{n} {}^{2^i}P_n$$

possible subset assignments. A brute force search through these subset assignments would be extremely inefficient.

Kameda and Weiner's solution to this problem and was a procedure to find at least one subset assignment which would result in an equivalent NFA. The procedure is based on concepts discussed in the next section.

Mapping

As shown above a brute force search through all the possible subset assignments for the minimal DFA would be extremely inefficient. Kameda and Weiner solved this problem in their paper [11] by making use of the properties of DFAs and duals as used in Brzozowski's DFA minimization algorithm.

The algorithm makes use of a structure known as a states map. This structure makes use of the concept of nonempty states. We provide both definitions and then discuss an example afterwards.

Definition 4.27 (Nonempty State)
Let $M = (Q, \Sigma, \delta, Q_0, F)$ be a FA with $\mathrm{DFA}(M) = (P, \Sigma, \delta', p_0, F')$.

From definition 2.11 (see page 13) we have that $p \subseteq Q$ with $p \in P$. With $p' \in P$, p' is defined as empty if there is no state $q \in p'$, otherwise p' is defined as non-empty.

Definition 4.28 (States Map)
Let $M = (Q, \Sigma, \delta, Q_0, F)$ be a FA, $\mathcal{B} = \mathrm{DFA}(M) = (Q', \Sigma, \delta', Q'_0, F')$ and $\mathcal{C} = \mathrm{DFA}(\overleftarrow{M}) = (Q'', \Sigma, \delta'', Q''_0, F'')$ the dual of M.

The states map (SM) of M is a matrix which contains a row for each nonempty state of $\mathcal{B}$ and a column for every nonempty state of $\mathcal{C}$. Every (i, j) entry in the matrix contains $q'_i \cap q''_j$, and if $q'_i \cap q''_j = \emptyset$ the entry is left blank.

The elementary automaton matrix (EAM) of M replaces every nonblank entry in the SM by a 1 and every blank entry by a 0.

The SM and EAM highlights for us which states in $\mathcal{B}$ and $\mathcal{C}$ are non-disjoint subsets of the states of M.

[12] ${}^m P_n = \frac{m!}{(m-n)!}$.

Example 4.29 (States Map)

From example 4.26 (see page 58) we have $M = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, \{q_1\}, \{q_3\})$ *where* δ *can be described as*

δ	0	1
q_0	$\{q_0\}$	$\{q_2\}$
q_1	$\{q_0, q_1\}$	$\{q_2\}$
q_2	$\{q_1\}$	$\{q_2\}$

.

Then

$$\mathcal{B} = \mathrm{DFA}(M) = (\{q_0', q_1', q_2', q_3'\}, \{0,1\}, \delta', q_0', \{q_1'\}),$$

where δ' *can be described as*

δ'	0	1
q_0'	q_0'	q_1'
q_1'	q_2'	q_1'
q_2'	q_3'	q_1'
q_3'	q_3'	q_1'

,

and

$$\begin{aligned} q_0' &= \{q_0\} \\ q_1' &= \{q_2\} \\ q_2' &= \{q_1\} \\ q_3' &= \{q_0, q_1\} \end{aligned} \quad .$$

Also,

$$\mathcal{C} = \mathrm{DFA}(\overleftarrow{M}) = (\{q_\emptyset'', q_0'', q_1''\}, \{0,1\}, \delta'', q_0'', \{q_1''\})$$

where δ'' *can be described as*

δ''	0	1
q_0''	$q_\emptyset''$	q_1''
q_1''	q_1''	q_1''

,

and

$$\begin{aligned} q_\emptyset'' &= \emptyset \\ q_0'' &= \{q_2\} \\ q_1'' &= \{q_0, q_1, q_2\} \end{aligned} \quad .$$

Therefore, the SM and EAM of M *are given as:*

	q_0''	q_1''
q_0'		q_0
q_1'	q_2	q_2
q_2'		q_1
q_3'		q_0, q_1

,

	q_0''	q_1''
q_0'	0	1
q_1'	1	1
q_2'	0	1
q_3'	0	1

.

$\mathcal{C}$ is the DFA obtained through the subset construction of the dual of M. It follows that $\mathcal{C}$ has no redundant states and represents M with no equivalent states. Similarly $\mathcal{B}$ represents M with no redundant states. From these two DFAs Kameda and Weiner attempted to determine which states could be reduced from the original FA.

The following theorem from [11] defines how the EAM can be used to determine the equivalence of states.

Theorem 4.30

Let E *be the EAM of a FA* M *as in definition 4.28 we have that*

$$sc(q_i') = \bigcup_{j|e_{ij}=1} \{\overleftarrow{x} | x \in pr(q_j{}'')\}.$$

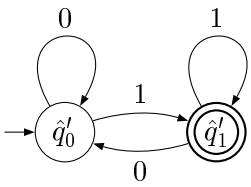

Figure 4.7: $\hat{\mathcal{B}}$ in example 4.32.

It follows that if two rows in an EAM E of a FA M have identical patterns of 0s and 1s they are equivalent. If the row representing q'_i covers[13] the row representing q'_j then $\text{sc}(q'_j) \subseteq \text{sc}(q'_i)$. Furthermore, if the row representing q'_i is equal to the union of a set of rows then q'_i is equivalent to the set of states represented by the union.

By working with $\overleftarrow{M}$, a similar theorem can be used to detect equivalent states in $\mathcal{C}$.

Combining the equivalent states in $\mathcal{B}$ and $\mathcal{C}$ produces reduced DFAs. Constructing matrices similar to the SM and EAM for the reduced DFA we obtain reduced states map and reduced automaton matrices, defined below:

Definition 4.31 (Reduced States Map)

Using the FAs from definition 4.28, the DFAs $\hat{\mathcal{B}} = (\hat{Q}', \Sigma, \hat{\delta}', \hat{Q}'_0, \hat{F}')$ *and* $\hat{\mathcal{C}} = (\hat{Q}'', \Sigma, \hat{\delta}'', \hat{Q}''_0, \hat{F}'')$ *can be constructed from* $\mathcal{B}$ *and* $\mathcal{C}$ *by combining the equivalent states obtained by comparing rows in the EAM.*

The reduced states map (RSM) of M *is a matrix which contains a row for each nonempty state of* $\hat{\mathcal{B}}$ *and a column for every nonempty state of* $\hat{\mathcal{C}}$*. Every* (i, j)*-th entry in the matrix contains* $\hat{q}'_i \cap \hat{q}''_j$*, if* $\hat{q}'_i \cap \hat{q}''_j = \emptyset$ *the entry is left blank.*

The reduced automaton matrix (RAM) of M *replaces every nonblank entry in the RSM by a* 1 *and every blank entry by a* 0*.*

Note 6 *When representing the SM and EAM we described each row and column by the state in the DFA. Similarly the RSM and RAM are described using the subsets of states from* $\mathcal{B}$ *and* $\mathcal{C}$*. It follows that if* $q \in \hat{Q}'$ *is the result of combining* $q_i, q_j \in Q'$ *then* q *is described as* $\{q_i, q_j\}$ *in the table. It may also be described as one of the combined states in square brackets, that is* $q = [q_i]$*.*

Example 4.32 (RAM)

[13] See definition A.13, page 171.

Continuing from example 4.29 we have the EAM of M as

	q_0''	q_1''
q_0'	0	1
q_1'	1	1
q_2'	0	1
q_3'	0	1

.

As the rows representing states q_0', q_1', q_3' are identical their states are equivalent. As no columns are identical no states in $\mathcal{C}$ are equivalent so that we obtain:

$$\hat{\mathcal{B}} = (\{\hat{q}_0', \hat{q}_1'\}, \{0,1\}, \hat{\delta}', \hat{q}_0', \{\hat{q}_1'\})$$

where $\hat{\delta}'$ can be described as (also given in figure 4.7)

$\hat{\delta}'$	0	1
$\hat{q}_0'$	$\hat{q}_0'$	$\hat{q}_1'$
$\hat{q}_1'$	$\hat{q}_0'$	$\hat{q}_1'$

,

where $\hat{q}_0' = \{q_0', q_2', q_3'\} = \{q_0, q_1\}$, $\hat{q}_1' = \{q_1'\} = \{q_2\}$

and

$$\hat{\mathcal{C}} = (\{\hat{q}_0'', \hat{q}_1'', \hat{q}_2''\}, \{0,1\}, \hat{\delta}'', \hat{q}_0'', \{\hat{q}_1''\})$$

where $\hat{\delta}''$ can be described as

$\hat{\delta}''$	0	1
$\hat{q}_0''$	$\hat{q}_\emptyset''$	$\hat{q}_1''$
$\hat{q}_1''$	$\hat{q}_1''$	$\hat{q}_1''$

,

where $\hat{q}_\emptyset'' = \emptyset = \emptyset$, $\hat{q}_0'' = \{q_0''\} = \{q_2\}$, $\hat{q}_1'' = \{q_1''\} = \{q_0, q_1, q_2\}$.

The RSM and RAM for M are given by:

	$\hat{q}_0''$	$\hat{q}_1''$
$\hat{q}_0'$		q_0, q_1
$\hat{q}_1'$	q_2	q_2

	$\hat{q}_0''$	$\hat{q}_1''$
$\hat{q}_0'$	0	1
$\hat{q}_1'$	1	1

.

RAMs can also be used to test for equivalence as follows from [11] as follows

Theorem 4.33 *Equivalent automata have a unique RAM, up to a permutation of rows and columns.*

The RAM of a FA M can be used to create a subset assignment to a minimal NFA. This is done by constructing a cover on the RAM.

A cover is defined on the concept of a grid defined as follows:

Definition 4.34 (Grid)

Given a RAM (or EAM), if all the entries at the intersections of a set of rows $\{q_{i_1}', \ldots, q_{i_a}'\}$ and a set of columns $\{q_{j_1}'', \ldots, q_{j_a}''\}$ are 1's then this set of 1's is said to form a grid.

A grid is represented by $g = (q'_{i_1}, \ldots, q'_{i_a}, q''_{j_1}, \ldots, q''_{j_b})$.

The grid g *is said to contain the pair* (q'_i, q''_j) *if* $i \in \{i_1, \ldots, i_a\}$ *and* $j \in \{j_1, \ldots, j_b\}$.

If a grid $g_1 \subseteq g_2$*, that is if* g_1 *contains* (q'_i, q''_j) *and* g_2 *contains* (q'_i, q''_j)*, then* g_2 *is said to contain* g_1*. That is, all the pairs contained in* g_1 *are contained in* g_2*.*

Definition 4.35 (Cover)

A cover is defined as a set of grids which contains every pair (q''_i, q''_j) *where the entry* (i, j) *in the RAM is a* 1*.*

The cover is defined as a minimum cover if it consists of the minimal number of grids.

By naming each grid in a given cover, a cover map (CM) is obtained by replacing the entries in the matrices containing 1*'s with the names of the grid.*

Note that the RSM is a CM of the RAM.

Lastly we need to be able to associate the CM with a subset assignment to be able to apply the intersection rule on the minimal DFA $\hat{\mathcal{B}}$. The subset assignment associated with the CM is constructed by mapping each state $\hat{q}'$ to a state with the label of each grid with an element in $\hat{q}'$s row in the CM.

That is:

Definition 4.36 (Subset Assignment)

Given a RAM associated with a FA M*, the DFA* $\hat{\mathcal{B}} = (\hat{Q}', \Sigma, \hat{\delta}', \hat{Q}'_0, \hat{F}')$*,* $\hat{\mathcal{C}} = (\hat{Q}'', \Sigma, \hat{\delta}'', \hat{Q}''_0, \hat{F}'')$ *and a CM associated with the RAM* $\mathcal{G}$ *we construct the subset assignment* $< \hat{Q}', f >$ *as follows:*

$$g \in f(\hat{q}') \text{ if } g \in \mathcal{G}, \text{ and } \exists \hat{q}'' \in \hat{Q}'', \text{ so that } g \text{ contains } (\hat{q}', \hat{q}'').$$

Example 4.37 (Cover Map)

Using the FAs defined for example 4.32 we have the RAM

	$\hat{q}''_0$	$\hat{q}''_1$
$\hat{q}'_0$	0	1
$\hat{q}'_1$	1	1

.

From the RAM we can construct three minimal CMs of the two grids α, β *as follows:*

	$\hat{q}''_0$	$\hat{q}''_1$
$\hat{q}'_0$		α
$\hat{q}'_1$	β	α, β

,

	$\hat{q}''_0$	$\hat{q}''_1$
$\hat{q}'_0$		α
$\hat{q}'_1$	β	β

and

	$\hat{q}''_0$	$\hat{q}''_1$
$\hat{q}'_0$		α
$\hat{q}'_1$	β	α

.

These three minimal CMs correspond to the following three subset assignments:

$$\begin{array}{lllllllll} f_0(\hat{q}'_0) & = & \alpha & , & f_1(\hat{q}'_0) & = & \alpha & \text{and} & f_2(\hat{q}'_0) = \alpha, \\ f_0(\hat{q}'_0) & = & \{\alpha, \beta\} & , & f_1(\hat{q}'_0) & = & \beta & \text{and} & f_2(\hat{q}'_0) = \{\alpha, \beta\}. \end{array}$$

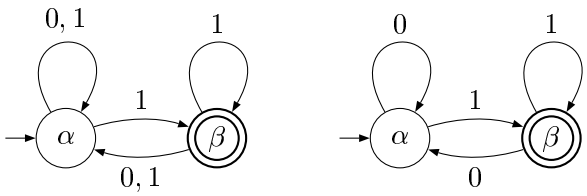

Figure 4.8: Example 4.36. Left: M_0, Right: M_1.

Note that $f_0 = f_2$ so we can discard f_2 further on.

Using the intersection rule on the remaining two subsets we obtain two FA $M_i = (\{\alpha, \beta\}, \{0, 1\}, \delta_i, \{\alpha\}, \{\beta\})$ with $i = 0, 1$ where δ_i can be described as(also shown in figure 4.8):

δ_0	0	1
α	α	$\{\alpha, \beta\}$
β	α	$\{\alpha, \beta\}$

δ_1	0	1
α	α	β
β	α	β

It follows that M_1 is a DFA with M_0 an NFA. Both FA have exactly two states and accept the same language $(0|1)^*1$ so that

$$M_0 \equiv M_1 \equiv M.$$

It is also easy to verify that M_0 and M_1 are minimal.

In the example the reconstruction was trivial as we dealt only with two states. Determining the reconstruction will not always be as simple a task for general NFAs.

Using the above constructions and concepts Kameda and Weiner proved the following theorem [11]:

Theorem 4.38 (Minimal NFA construction)
A FA $M_0 = I(\hat{\mathcal{Q}}', f, \hat{\mathcal{B}})$ is a minimal FA accepting $\mathcal{L}(\hat{\mathcal{B}})$ if f is associated with a minimal CM of the RAM associated with $\hat{\mathcal{B}}$.

Using this theorem we can attempt to construct minimal NFAs equivalent to a given NFA by constructing the RAM and minimal DFA equivalent to the given NFA. This is done by generating CMs starting with the minimal covers and using the intersection rule attempt to reconstruct an equivalent NFA. If no such NFA is found the CMs can be constructed with increasing number of grids. The number of CMs possible is limited, and the CMs can consist of at most $\min(|\hat{Q}'|, |\hat{Q}''|)$ number of grids.

From the theorem we have that the algorithm guarantees results. However, as the subset construction algorithm is used in the minimization algorithm, and multiple FAs need to be stored at once, the space and worst case computational complexity exceeds $O(2^n)$ where $n = |Q|$. This means

that the algorithm as with all NFA minimization algorithms would be inefficient to run. NFA reduction algorithms circumvent this problem by not determining every possible reducible. Instead they use simplified tests to determine reducible states.

4.3.2 NFA reduction

Although Kameda and Weiner's algorithm will generate the minimal NFA, its worst case computational complexity makes it an inefficient solution to the problem of state minimization of NFAs. In fact as shown in [10] the problem of NFA state minimization problem is NP hard. This means that any NFA minimization algorithm would be inefficient. On the other hand, NFA reduction algorithms based on finding and reducing only specific types of reducible states can run in polynomial time.

Recently Ilie and Yu [8], Champarnaurd and Coulon [3] and then Ilie, Navarro and Yu [9] released NFA reduction algorithms based on equivalences and preorders which we will discuss later in this section. Ilie, Navarro and Yu [9] presented algorithms which have worst case computational complexity of $\mathcal{O}(m \log n)$ based on equivalence relations and $\mathcal{O}(mn)$ based on preorders where m is the number of transitions and n is the number of states.

In this section we describe two reduction algorithms, the first by Ilie and Yu [8] based on equivalences and the second by Champarnaurd and Coulon [3] based on preorders. We do not provide the algorithms themselves as we are more interested in the applications of these algorithms for use with $\oplus$-FA than with the algorithms themselves. Instead we describe the tests used to determine which states may be reduced by using equivalences and preorders.

Equivalence Reduction

The first NFA reduction algorithm reduces the NFA by reducing only equivalent states as defined in definition 4.4 (page 31). The algorithm by Champarnaud and Coulon [3] ran in polynomial time which was improved by Ilie, Navarro and Yu [9] through the use of a search algorithm. We present the definitions and equivalence tests needed for the reduction algorithm but do not discuss the search algorithm used[14].

In corollary 4.3 (page 31), we have shown that computing and comparing two or more states predecessors with each other is NP hard. For this reason a simplified equivalence test is used similar to Hopcroft's DFA minimization algorithm [7]. The equivalence test used is based on a sequence of definitions of equivalence as follows:

[14]The search algorithm was left open by Champarnaud and Coulon so that the reader may make use of the search algorithm they feel most comfortable with.

We denote two equivalent states q_i, q_j in a FA $M = (Q, \Sigma, \delta, Q_0, F)$ where $q_i, q_j \in Q$ as

$$q_i \equiv q_j.$$

Another equivalence relation $\equiv_0$ is then constructed so that $\equiv_0$ is contained by $\equiv$, that is

$$q_i \equiv_0 q_j \;\Rightarrow\; q_i \equiv q_j.$$

Therefore, if $q_i \equiv_0 q_j$ then $q_i \equiv q_j$ and the states $q_i,\ q_j$ can be merged as.

We start by defining $\equiv_0$ as the coarsest equivalence relation on Q, such that

Definition 4.39 (Coarse Equivalence)
In a FA $M = (Q, \Sigma, \delta, Q_0, F)$ with $q_i, q_j \in Q$, $q_i \equiv_0 q_j$ if

- $q_i \in F \Leftrightarrow q_j \in F$, *and*
- *for every $\sigma \in \Sigma$ and for every $q_i' \in \delta(q_i, \sigma)$ there exists $q_j' \in \delta(q_j, \sigma)$ so that*

$$q_i' \equiv_0 q_j'.$$

In this way an NFA can be divided into equivalence classes similar to Hopcroft's DFA minimization algorithm discussed in section 4.2.2. These equivalence classes are then combined to form a reduced NFA.

Note 7 *It is important to note that definitions 4.11 (see page 42) and 4.39 above differ in their tests. In the case of the DFA exactly one transition is checked for every alphabet symbol. However, for the NFA an equivalent state is compared for each transition on every alphabet symbol.*

This means that although an algorithm for computing such equivalence relations will be quite similar to Hopcroft's algorithm, the worst case computational complexity is increased due to the number of transitions checked.

It follows that an algorithm based on definition 4.39 will only reduce equivalent states. We also know that in the dual of an NFA the predecessors become the successors of the states. By using the dual we can compute redundancy in the same way as equivalence above.

That is

Corollary 4.40 (Coarse Redundancy)
In a FA $M = (Q, \Sigma, \delta, Q_0, F)$ with dual $\overleftarrow{M} = (Q, \Sigma, \overleftarrow{\delta}, F, Q_0)$, as defined in definition 4.17, with $q_i, q_j \in Q$, q_i and q_j are redundant if

- $q_i \in Q_0 \Leftrightarrow q_j \in Q_0$, *and*
- *for every $\sigma \in \Sigma$ and for every $q_i' \in \overleftarrow{\delta}(q_i, \sigma)$ there exists $q_j' \in \overleftarrow{\delta}(q_j, \sigma)$ so that*

$$q_i' \text{ and } q_j' \text{ are redundant.}$$

By making use the corollary and definition 4.39 we can generate algorithms to detect and remove equivalent and redundant states from an NFA. However, final-useless, initial-useless, and contained states are not reduced. In fact not all equivalent or redundant states are reduced as $\equiv_0 \Rightarrow \equiv$ so that $\equiv \not\Rightarrow \equiv_0$.

Preorder Reduction

We have discussed the idea of reducing an NFA through the use of the approximation of equivalence through coarse equivalence. Champarnaud and Coulon [3] extended the idea of equivalence relations to preorders which are simpler to determine.

In this section we have mentioned preorders without formally defining them. Preorders are a measure of comparison between predecessors and successors of states. This is similar to subsets for sets.

Definition 4.41 (Preorders) *In a FA $M = (Q, \Sigma, \delta, Q_0, F)$ with $q_i, q_j \in Q$, we define the following two preorders $\overleftarrow{\subseteq}$ and $\overrightarrow{\subseteq}$ so that*

$q_i \overleftarrow{\subseteq} q_j$ *if and only if* $pr(q_i) \subseteq pr(q_j)$, *and*

$q_i \overrightarrow{\subseteq} q_j$ *if and only if* $sc(q_i) \subseteq sc(q_j)$.

Using the definition of preorders we have that the following reducible states can be defined as follows:

Corollary 4.42 *In a FA $M = (Q, \Sigma, \delta, Q_0, F)$ with $q_i, q_j \in Q$, we have that*

- q_i *and* q_j *are redundant if*

$$q_i \overleftarrow{\subseteq} q_j, \text{ and } q_j \overleftarrow{\subseteq} q_i,$$

- q_i *and* q_j *are equivalent if*

$$q_i \overrightarrow{\subseteq} q_j, \text{ and } q_j \overrightarrow{\subseteq} q_i, \text{ and}$$

- q_i *is contained by* q_j *if*

$$q_i \overrightarrow{\subseteq} q_j, \text{ and } q_i \overleftarrow{\subseteq} q_j.$$

That is, states are redundant with each other if both of their predecessors form subsets of each other. Similarly states are equivalent if their successors form subsets of each other. Finally a state is contained if both its predecessor and successor form subset of another states predecessor and successor.

As with equivalence reduction where an approximation to the equivalence relation, namely coarse equivalence, was constructed to simplify the reduction process. Champarnaud and Coulon [3] created approximations to preorders namely $\overleftarrow{\subseteq}_0$ and $\overrightarrow{\subseteq}_0$ where

$$\overleftarrow{\subseteq}_0 \Rightarrow \overleftarrow{\subseteq} \text{ and } \overrightarrow{\subseteq}_0 \Rightarrow \overrightarrow{\subseteq}.$$

However, these initial preorders were not used by Champarnaud and Coulon. Instead they made use of another approximation called the first order approximation of inequalities.

That is

Definition 4.43 (First Order)

In a FA $M = (Q, \Sigma, \delta, Q_0, F)$ *with* $q_i, q_j \in Q$, $q_i \overrightarrow{\subseteq}_1 q_j$ *if*

- $q_i \in F \Leftrightarrow q_j \in F$, *and*
- *for every* $\sigma \in \Sigma$ *and for every* $q_i' \in \delta(q_i, \sigma)$ *there exists* $q_j' \in \delta(q_j, \sigma)$ *so that*

$$q_i' \overrightarrow{\subseteq}_1 q_j'.$$

Similarly for $\overleftarrow{\subseteq}_1$.

That is a first order approximation of inequalities is where a state $q_i \overrightarrow{\subseteq}_1 q_j$ if either both are final or non final and every child state of q_i is a preorder of a child in q_j.

By using this definition to test for the preorders we can determine states which are redundant, equivalent and contained, without referencing the dual of the NFA. Although not described here the algorithm constructed by Champarnaud and Coulon makes use of matrices which represent transitions to calculate preorders.

A non optimized implementation of an algorithm based on preorders runs with worst case computational complexity of about $\mathcal{O}(mn^3)$. The optimized algorithm by Champarnaud and Coulon improved the efficiency of the algorithm in [3] to $\mathcal{O}(mn)$. It is of interest to note that in specific circumstances including the use of parallelization techniques for matrices the non-optimized algorithm may run faster than the optimized version.

Champarnaud and Coulon went on to examine more complex approximations to preorders for better results. However, the worst case computational complexity resulting from increasing the approximation grows exponentially. This can be expected since the calculation of complete preorders would be NP-hard due to the nature of NFA minimization. The results from the first order approximations to the preorders results in a dramatic reduction of states and transitions. This is a satisfactory result for applications where it is not the minimal NFA which is necessary to compute but the minimal

DFA. Such reduction algorithms can also be used as precursors to the more complex NFA minimization algorithms such as Kameda and Weiner which would decrease their computational time.

4.4 Bideterminism

In the previous two sections we have discussed minimization and reduction algorithms for both DFAs and NFAs. From theorem 4.21 we know that the unique minimal DFA gives a range bound for the minimal NFAs. It is interesting to note exactly when a minimal NFA has the same number of states as the minimal DFA. Tamm and Ukkonen [16] investigated a specialized form of DFAs, namely the bideterministic finite automata.

We start by defining these automata, after which we discuss the properties of such automata.

Definition 4.44 *Bideterminism*

*A **bideterministic finite automaton**, $M = (Q, \Sigma, \delta, Q_0, F)$ is a DFA whose dual, $\overleftarrow{M}$ is also a DFA, that is:*

- *There is only one start and final state,*

$$|Q_0| = |F| = 1, \text{ and}$$

- *both the transition function and its dual have at most one transition for each state on each alphabet symbol, that is for $\sigma \in \Sigma$ and $q \in Q$:*

$$|\delta(q, \sigma)| \leq 1 \ , \ |\overleftarrow{\delta}(q, \sigma)| \leq 1.$$

This means that each state has at most one transition to and from it for each alphabet symbol.

We can further define a complete bideterministic finite automaton as having exactly one transition to and from each state for each alphabet symbol.

From [16] we have that:

Theorem 4.45 (Minimality)

A complete bideterministic finite automaton with no initial or final-useless states is both the minimal DFA and a minimal NFA.

This means that any bideterministic finite automaton only needs to be checked for initial and final-useless states to ensure that it is minimal.

Furthermore, Tamm and Ukkonen [16] provided the following insight into bideterminism, that is:

Theorem 4.46 (Uniqueness)
A minimal bideterministic finite automaton, $M = (Q, \Sigma, \delta, Q_0, F)$, is unique among NFAs. That is any NFAs, $M' = (Q', \Sigma, \delta', Q_0', F')$ where $M \equiv M'$ and which are not isomorphic[15] *to M have the property that*

$$|Q| < |Q'|.$$

Bideterministic finite automata are of interest to us because theorems 4.45 and 4.46 hold not only for NFAs but also for all the $*$-FA [16].

In this chapter we have discussed various FA minimization algorithms. The purpose of this chapter was to gain some background knowledge into current minimization algorithms and insight into the worst case computational complexity involved in such minimization algorithms. We discussed three main branches of minimization algorithms, namely the language construction, state comparison and transformation reconstruction. We also noted that in state comparison the problem of minimization can be simplified to reduction to increase the speed of such algorithms.

The algorithms were based on defined reducible states which allow us to target specific areas of the FA where minimization can take place. In the next section we discuss the minimization of $\oplus$-FA. We begin by discussing reducible states and then move onto simplified and general minimization algorithms.

[15] As described in A.12.

Chapter 5

Minimization of $\oplus$-FA

FAs have existed in the literature since the early 1930s [15] and as such many applications have been developed using these traditional FAs. To compensate for the large FAs being used by many of these applications, minimization and reduction algorithms have been constructed to attempt to reduce the number of states efficiently.

More recently, applications have been developed for $\oplus$-FAs such as hashing [14] and random number generation [18]. As with traditional NFAs, if large $\oplus$-FAs are to be used in applications they would need to be handled efficiently. This includes having efficient methods to store and run $\oplus$-FAs, as well as being able to efficiently reduce the number of states. In this chapter, we attempt to construct $\oplus$-FA minimization and reduction algorithms to handle future concerns of such large $\oplus$-FA applications.

As little research has been done into the minimization of $\oplus$-FAs, we are faced with a blank slate to develop such algorithms. As such, the $\oplus$-FA minimization algorithms we construct are based on both properties of the $\oplus$-FAs and traditional FA minimization algorithms. By examining the minimization algorithms from chapter 4 we can gain a better understanding of the functioning and minimization of $\oplus$-FAs.

The traditional FA minimization algorithms are based on the reducible states of these FAs. However, as the $\oplus$-operator does not react with transitions in the same way as the union operator does, the reducible states as defined for traditional FAs are not always applicable to $\oplus$-FAs. Similarly the calculation of these reducible states is made more complex due to the interaction of these states.

In this chapter we distinguish the minimization of two forms of $\oplus$-FAs. We begin with unary $\oplus$-FAs, that is, $\oplus$-FAs with exactly one alphabet symbol. We develop a minimization algorithm for unary $\oplus$-FAs based on their relationship with LFSRs [14]. In section 5.2 we then discuss the minimization of the general $\oplus$-FAs. We begin by examining the reducible states of the traditional NFAs in relation to $\oplus$-FAs. Using these reducible states we

examine how these reducible states can be used to construct minimization algorithms of the types: Language reconstruction (section 3.2.2); transformation construction (section 3.2.3); and state comparison (section 3.2.3), with varying success. In each case we describe the idea behind the minimization algorithm as well as provide examples and problems encountered.

5.1 Unary $\oplus$-FAs

We begin our discussion of the minimization of $\oplus$-FAs with a simplified form, namely the unary $\oplus$-FA. A unary $\oplus$-FA is simply a $\oplus$-FA with exactly one alphabet symbol. During our discussion, we assume without loss of generality that the alphabet of a unary $\oplus$-FA is the set $\{a\}$.

The minimization algorithm we present here can take as input any unary $\oplus$-FA with multiple final states and produces as output a unary $\oplus$-FA with exactly one final state. It is important to note that the produced unary $\oplus$-FA is minimal amongst unary $\oplus$-FAs with one final state, but it is not necessarily minimal among unary $\oplus$-FAs with multiple final states. We show later that the algorithm can be adjusted to produce a unary $\oplus$-FA with multiple final states, however, with a proportional increase in computational complexity. Furthermore, in the following chapter, we shall discuss how the produced unary $\oplus$-FA with one final state can be used to create an efficient format to store unary $\oplus$-FAs.

We shall use the following running example to illustrate our minimization algorithm:

Example 5.1 (Original Unary $\oplus$-FA)
Let M be a $\oplus$-FA defined by

$$M = (\{q_1, q_2, q_3, q_4, q_5\}, \{a\}, \delta, \{q_1\}, \{q_5\}, \oplus)$$

with δ given by (see also figure 5.1)

δ	a
q_1	$\{q_2, q_4\}$
q_2	$\{q_5\}$
q_3	$\{q_1, q_2, q_4\}$
q_4	$\{q_2\}$
q_5	$\{q_3, q_4\}$

.

5.1.1 Preliminaries

In this section we describe certain concepts, such as the final state sequence and the characteristic polynomial, which we use for the minimization of unary $\oplus$-FAs. We also discuss how these sequences and polynomials can be generated.

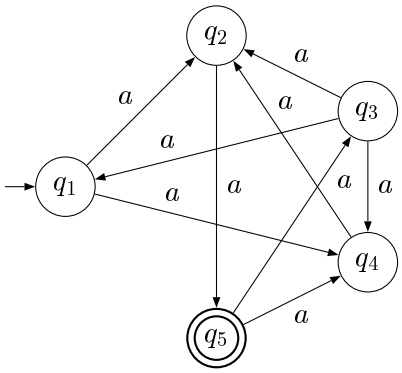

Figure 5.1: Example 5.1

Before discussing these concepts we begin with discussing the cyclic nature of unary $\oplus$-FAs. This cyclic nature is demonstrated in the so-called *state sequence* of the unary $\oplus$-FA.

A state sequence is simply the sequence of active state sets produced, given a sequence of alphabet symbols as input. However, as a unary $\oplus$-FA has exactly one alphabet symbol, the input sequence can consist only of that alphabet symbol. As such a unary $\oplus$-FA can only ever have one state sequence for an input sequence of a certain length.

As a $\oplus$-FA consists of a finite number of states, n, the number of possible state sets is also finite, 2^n, by the subset construction. Also, given an active state set, the following active state set will always be predetermined on a given alphabet symbol. This means that if a state set occurs twice in the state sequence of a unary $\oplus$-FA, the state sets between the two occurrences of the repeated state sets will be continuously repeated in the state sequence. These repeated state sets are the so called cycle in the sequence.

The state sequence can be generated by recording the initial state set and then repeatedly calculating the following active state set on input a until a state set is repeated. However, this can be seen as on-the-fly subset construction, as the resultant state sequence is simply the states which occur in the equivalent DFA. This implies that the computation of the state sequence has a worst case computational complexity of $\mathcal{O}(2^n)$.

With this in mind we formally define the state sequence of a unary $\oplus$-FA as follows:

Definition 5.2 (State Sequence)
The ***state sequence*** $S = S_0, S_1, \ldots$ *of a unary* $\oplus$*-FA* $M = (Q, \{a\}, \delta, Q_0, F, \oplus)$ *is defined as*

$$S_i \;=\; \delta(Q_0, a^i).$$

Note 8 *Note that we when we write a state sequence we write only the subsequence $S_0, S_1, \ldots, S_i$ where S_i is the first repeated state set. That is*

$$S_i \ = \ S_j, \ for \ some \ j < i.$$

The sequence S is denoted as

$$S_0, \ S_1, \ \ldots, \ S_j, \ \ldots, \ S_i.$$

Furthermore we refer to the state sequence $S_j, S_{j+1}, \ldots, S_{i-1}$ as the cycle of the state sequence.

In the running example we have:

Example 5.3 (State Sequence)
The state sequence generated by M in example 5.1 (page 74) is

$$\{q_1\}, \ \{q_2, q_4\}, \ \{q_2, q_5\}, \ \{q_3, q_4, q_5\}, \ \{q_1, q_3\}, \ \{q_1\}.$$

As the active state set $\{q_1\}$ occurs twice we know that the state sequence

$$\{q_1\}, \ \{q_2, q_4\}, \ \{q_2, q_5\}, \ \{q_3, q_4, q_5\}, \ \{q_1, q_3\},$$

is the cycle as it would be repeated continuously.

It is shown in [18] that all unary $\oplus$-FAs will have such repeating state sequences.

To construct a reduced unary $\oplus$-FA we attempt to generate a unary $\oplus$-FA with fewer states whose state sequence emulates the original state sequence. By emulating a state sequence we mean a state sequence with state sets containing a final state, only where the state set from the original sequence also contains a final state. We begin by discussing the nature of the state sequence.

A state sequence is described by the active state sets up to and including the first repeated state set. The sequence is further described according to the length of the cycle and the *preamble*. The preamble of a sequence is the subsequence consisting of the state sets in the sequence but not in the cycle. That is, in a sequence, S, with the first repeated state set $S_i = S_j$ with $j < i$, the preamble consists of the states $S_0, S_1, \ldots, S_{j-1}$.

As mentioned briefly in the introduction, unary $\oplus$-FA are related to linear feedback shift registers (LFSRs) [5]. This relationship is manifested through the state sequence. A LFSRs state sequence can be described by the so-called characteristic matrix and a set of initial states.

In the case of $\oplus$-FAs, the characteristic matrix is simply a representation of the transition function of the $\oplus$-FA. As such the transition function is encoded into a binary matrix as follows:

Definition 5.4 (Characteristic Matrix)
Given a unary $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ the characteristic matrix is a binary $|Q| \times |Q|$ matrix $\mathbf{A}$ where the entries of the matrix a_{ij} are given by

$$a_{ij} = \begin{cases} 1 & \text{if } q_i \in \delta(q_j, a), \\ 0 & \text{otherwise.} \end{cases}$$

Each row in the transition table is translated into a column of the characteristic matrix, with entries of 1 in those rows which contain transitions and 0's otherwise. As such entries of 1 represent the transitions.

The characteristic matrix allows us to examine unary $\oplus$-FAs using matrix calculations in the Galois field[1] $GF(2)$ as shown in [17]. By encoding the initial state set as a binary vector, the following active state set can be calculated through the $GF(2)$ multiplication of the vector by the characteristic matrix. Further multiplications by the matrix result in the active state sets generated in the same way as the state sequence.

The *characteristic polynomial* is calculated from the characteristic matrix as follows [17]:

Definition 5.5 (Characteristic Polynomial)
Given a characteristic matrix $\mathbf{A}$ as in definition 5.4, the characteristic polynomial $c(X)$ is defined as [2]

$$c(X) \;=\; det(\mathbf{A} - \mathbf{I}X).$$

Here $c(X) = X^n + c_{n-1}X^{n-1} + \ldots + c_0X^0$ where $c_i = 0$ or 1 and $c(X)$ is said to have degree n.

It follows directly from definition 5.5 that as every entry on the main diagonal[3] of the matrix $\mathbf{A} - \mathbf{I}X$ is either X or $X - 1$, a unary $\oplus$-FA with n states will have a characteristic polynomial of degree $n + 1$.

Thus the problem of finding the minimal $\oplus$-FA is equivalent to finding a characteristic polynomial of smallest degree which generates the given unary $\oplus$-FA's state sequence.

[1] See A.14, page 171. The reader may refer to [5] for an exposition on linear field theory and in particular $GF(2)$.

[2] The determinant or $det(\mathbf{A})$ of a matrix, $\mathbf{A}$, is defined in [5].

[3] The main diagonal of the $n \times n$ matrix $\mathbf{A}$ consists of all the entries a_{ij} where $i = j$, with $0 \leq i, j < n$.

In the running example we have:

Example 5.6 (Characteristic Matrix and Polynomial)
Using the unary $\oplus$-FA, M, from example 5.1 (page 74), the characteristic matrix $\mathbf{A}$ *of M is given by*

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix},$$

with the characteristic polynomial $c(X) = det(\mathbf{A} - \mathbf{I}X) = X^5 + 1$ of degree 5. If we encode the initial state set as the row vector $y = [1\ 0\ 0\ 0\ 0]$, we get

$$\begin{aligned} \mathbf{A^1}y^T &= [0\ 1\ 0\ 1\ 0], \\ \mathbf{A^2}y^T &= [0\ 1\ 0\ 0\ 1], \\ \mathbf{A^3}y^T &= [0\ 0\ 1\ 1\ 1], \\ \mathbf{A^4}y^T &= [1\ 0\ 1\ 0\ 0], \text{ and} \\ \mathbf{A^5}y^T &= [1\ 0\ 0\ 0\ 0]. \end{aligned}$$

It is easy to verify that the resultant row vectors correspond to the state sequence generated in example 5.3 (page 76). For example we see that $[0\ 1\ 0\ 1\ 0]$ corresponds to the state set $\{q_2, q_4\}$ and that $[0\ 1\ 0\ 0\ 1]$ corresponds to the state set $S_2 = \{q_2, q_5\}$.

To this end we combine the state sequence with the final states to form the final state sequence as follows:

Definition 5.7 (Final State Sequence)
Let M be a unary $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with state sequence $S = S_0, S_1, \ldots$. The ***final state sequence*** *$f = f_0, f_1, \ldots$ of M is defined as*

$$f_i = \begin{cases} 1 & \text{if } S_i \cap F \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

That is, given a state sequence $S = S_0, S_1, \ldots, S_i$ the final state sequence $f = f_0, f_1, \ldots, f_{i-1}$ is constructed so that if the state set S_j contains a final state then $f_j = 1$, otherwise $f_j = 0$. The final state sequence contains a cycle of 0 and 1s of the same length as the state sequence. However, where we write the state sequence up to and including the first repeated state set, we write the final state sequence up to but not including the first repeated state sequence.

The resultant binary final state sequence represents the language that the $\oplus$-FA accepts. A word of length m is accepted by a $\oplus$-FA with final

state sequence f if and only if $f_m = 1$. Thus any equivalent $\oplus$-FA must have the same final state sequence. In fact given only the final state sequence, the algorithm we present will generate a minimal unary $\oplus$-FA with one final state which accepts the language of the final state sequence.

Example 5.8 (Final State Sequence)
The final state sequence f of the unary $\oplus$-FA M in example 5.1 (page 74), with reference to examples 5.3 and 5.6 (pages 76 and 78), is:

$$f \;=\; 0,\ 0,\ 1,\ 1,\ 0,$$

with a cycle of length 5.

Note 9 *It is important to note here that although we refer to the characteristic polynomial as a defining characteristic of a unary $\oplus$-FA, it is not sufficient to describe the language completely. That is, different languages may be accepted by unary $\oplus$-FAs which have the same characteristic polynomial.*

For example, by changing the initial state set in example 5.1 (page 74) to $\{q_2\}$ we have the following state sequence

$$\{q_2\},\ \{q_5\},\ \{q_3, q_4\},\ \{q_1, q_4\},\ \{q_4\},\ \{q_2\},$$

this has the final state sequence,

$$0,\ 1,\ 0,\ 0,\ 0,$$

with a cycle of length 5.

Although the characteristic polynomial remains the same the language accepted differs. For this reason it is important to take note of the final state sequence when minimizing.

We have discussed how to describe a unary $\oplus$-FA by the use of state sequences and final state sequences and how to determine the final state sequences of a given unary $\oplus$-FA. This concludes the first step of the minimization algorithm. In the second step we will show how the final state sequence can be used to calculate the characteristic polynomial of smallest degree. The third and final step uses both the minimal characteristic polynomial and final state sequence to construct a minimal unary $\oplus$-FA with exactly one final state.

5.1.2 Berlekamp-Massey Algorithm

The Berlekamp-Massey algorithm [12] is a well known algorithm in the field of cryptology, used to break and generate stream ciphers [1]. The algorithm generates the minimal characteristic polynomial for a LFSR generating the given binary sequence of finite length. We use a modification of

this algorithm to find the characteristic polynomial of smallest degree which generates a given final state sequence.

The Berlekamp-Massey (BM) algorithm works by considering each bit s_k of an inputted binary sequence s in sequence order. It constructs the polynomial $c(X)$ which will generate the sequence $s_0 s_1 s_2 \ldots s_k$ based on the minimal polynomial which generates the sequence $s_0 s_1 s_2 \ldots s_{k-1}$. The algorithm does this by first checking whether the polynomial which generates the previous sequence also generates the new sequence. This is done by using the result of Berlekamp that the polynomial $c(X)$ generates the sequence $s_0 s_1 s_2 \ldots s_k$ if and only if $\sum_{i=0}^{k-1} c_i s_{k-i} = 0$. As such the algorithm compares each next bit of the sequence to the current polynomial and if it is cannot be generated by $c(X)$, $c(X)$ is recalculated. The algorithm continues until all the bits in the sequence have been processed.

For use in our algorithm we have modified the original Berlekamp-Massey algorithm. This was done to allow for the incorporation of cycles. That is, if run on a sequence $s = s_0 s_1 \ldots s_n$ with a cycle of length $l \leq n + 1$ it will produce the polynomial which will generate the sequence with cycle. This is done by running the algorithm on the repeated sequence until the characteristic polynomial is not modified for the length of the cycle.[4]

Before presenting the algorithm we mention some variables used by the algorithm:

- t records the current time step of the algorithm,
- $T(X)$ and $B(X)$ are temporary polynomials used for calculation purposes,
- L is the degree of $c(X)$, and
- m shows where L was last modified.

Algorithm 5.9 (Modified Berlekamp-Massey)

Given an input sequence $s = s_0, s_1, \ldots, s_{n-1}$ of length[5] n with cycle length l, the algorithm outputs the minimal polynomial $c(X)$ which generates the sequence.

(1) Initialization: $c(X) \leftarrow 1$, $T(X) \leftarrow 0$, $L \leftarrow 0$, $m \leftarrow -1$, $B(X) \leftarrow 1$, $t \leftarrow 0$, $k \leftarrow 0$.

(2) While $(k < n)$ *do*

(a) $d \leftarrow (s_t + \sum_{i=1}^{L} s_{t-i} c_i) (\text{mod } 2)$

[4] It is also important to note that we use reverse characteristic polynomials to what is used in [12]. That is, where $c(X) = c_n X^n + c_{n-1} X^{n-1} + \ldots + c_1 X + 1$ is generated in [12], we generate $c(X) = X^n + c_1 X^{n-1} + \ldots + c_{n-1} X + c_n$.

[5] Although we say the sequence has length n, it actually has infinite length due to the cycle. If the algorithm references the bit s_i with $i \geq n$ then $s_i = s_{i-l}$.

(b) if $d = 1$ *then do*

$T(X) \leftarrow c(X)$

If $2L \leq t$ *then*

$c(X) \leftarrow c(X) \cdot X^{t+1-2L} + B(X)$

$L \leftarrow t + 1 - L$, $m \leftarrow t$, $B(X) \leftarrow T(X)$

$k \leftarrow -1$

else

$c(X) \leftarrow c(X) + B(X) \cdot X^{2L-t-1}$

(c) $k \leftarrow k + 1$, $t \leftarrow t + 1$

(3) Return $c(X)$

In the running example we have:

Example 5.10

Taking the final state sequence $f = 00110$ *with a full cycle length of* 5 *from example 5.8 , the modified Berlekamp-Massey algorithm 5.9 yields the following results:*

s	d	$T(X)$	$c(X)$	L	m	$B(X)$	N
–	–	0	1	0	-1	1	0
0	0	0	1	0	-1	1	1
0	0	0	1	0	-1	1	2
1	1	1	$1 + X^3$	3	2	1	3
1	1	$1 + X^3$	$1 + X^2 + X^3$	3	2	1	4
0	1	$1 + X^2 + X^3$	$1 + X^1 + X^2 + X^3$	3	2	1	5
0	0	$1 + X^2 + X^3$	$1 + X^1 + X^2 + X^3$	3	2	1	6
0	1	$1 + X^1 + X^2 + X^3$	$1 + X^1 + X^2 + X^3 + X^4$	4	6	$1 + X^1 + X^2 + X^3$	7
1	0	$1 + X^1 + X^2 + X^3$	$1 + X^1 + X^2 + X^3 + X^4$	4	6	$1 + X^1 + X^2 + X^3$	8
1	0	$1 + X^1 + X^2 + X^3$	$1 + X^1 + X^2 + X^3 + X^4$	4	6	$1 + X^1 + X^2 + X^3$	9
0	0	$1 + X^1 + X^2 + X^3$	$1 + X^1 + X^2 + X^3 + X^4$	4	6	$1 + X^1 + X^2 + X^3$	10
0	0	$1 + X^1 + X^2 + X^3$	$1 + X^1 + X^2 + X^3 + X^4$	4	6	$1 + X^1 + X^2 + X^3$	11
0	0	$1 + X^1 + X^2 + X^3$	$1 + X^1 + X^2 + X^3 + X^4$	4	6	$1 + X^1 + X^2 + X^3$	12

Thus, the characteristic polynomial returned from the algorithm is $c'(X) = 1 + X^1 + X^2 + X^3 + X^4$.

In this section we discussed the modified Berlekamp-Massey algorithm which constructs the minimal characteristic polynomial for a specific final state sequence. In the next section we discuss an algorithm to construct a unary $\oplus$-FA with one final state from the generated characteristic polynomial and final state sequence.

5.1.3 Construction of the $\oplus$-FA

In the previous section we discussed how to construct the minimal characteristic polynomial from the generated final state sequence in the first

step of the algorithm. In this section we discuss how to combine these two properties to construct a unary $\oplus$-FA with the desired properties.

We wish to construct a unary $\oplus$-FA $M' = (Q', \Sigma, \delta', Q'_0, F', \oplus)$ from a final state sequence f and a characteristic polynomial $c'(X)$. We can start by setting the alphabet to $\{a\}$ without loss of generality. The next step is to construct the state set Q' on which the other properties are dependant.

We discussed earlier that the number of states of the LFSR [18] is equal to the degree of the characteristic polynomial of the LFSR [5]. Thus with $c'(X)$ of degree n', the state set of M' can be defined as $Q' = \{q'_0, q'_1, \ldots, q'_{n'-1}\}$.

In order to construct M' we still need to construct the transition function δ', the initial state set Q'_0 and the final state set F'. Using the state set Q' and the characteristic polynomial it is in principle possible to construct different characteristic matrices $\mathbf{A}$ which represent the transition function. Although multiple such $\mathbf{A}$'s exist, it would be preferable to have a simple algorithm to construct a generic $\mathbf{A}$. One such generic form is known as the normal form[6] of characteristic matrices [5], and it is this normal form which we will use to construct the unary $\oplus$-FA.

The normal form is given in [17] as:

Definition 5.11 (Characteristic Matrix Normal Form)

Given the state set $Q' = \{q'_0, q'_1, \ldots, q'_{n'-1}\}$ *and characteristic polynomial* $c(X) = c_{n'}X^{n'} + c_{n'-1}X^{n'-1} + \ldots + c_1X^1 + c_0$, *the normal form of* $\mathbf{A}$ *is given by*

$$\begin{bmatrix} 0 & 0 & \ldots & 0 & c_0 \\ 1 & 0 & \ldots & 0 & c_1 \\ 0 & 1 & \ldots & 0 & c_2 \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & c_{n'-2} \\ 0 & 0 & \ldots & 1 & c_{n'-1} \end{bmatrix}.$$

By using the algorithm for the construction of a characteristic matrix from a transition function in definition 5.4 (page 77) we can construct the corresponding transition function δ' with Q' and $c'(X)$ as follows:

$$\begin{aligned} \delta'(q'_i, a) &= \{q'_{i+1}\}, \quad \text{for } 0 \leq i < n' - 1 \\ \delta'(q'_{n'-1}, a) &= \{q'_j \mid c_j = 1, \text{ for } 0 \leq j \leq n' - 1\}. \end{aligned}$$

Next we assign the final state. We do this by looking at LFSRs. Without going into detail, the output state in the LFSR feeds signals back to the non output states similar to the transitions from $q'_{n'-1}$ to the other states. Emulating the LFSR we assign $q'_{n'-1}$ as the only final state, that is, $F' = \{q'_{n'-1}\}$.

[6] Any matrix $\mathbf{A}$ in GF(2) has a normal form, namely $\mathbf{PAP}^{-1}$, where $\mathbf{P}$ is a matrix similar to $\mathbf{A}$. A square matrix $\mathbf{P}$ is defined as similar to square matrix $\mathbf{A}$ if there exists a square nonsingular matrix $\mathbf{X}$ so that $\mathbf{P} = \mathbf{X}^{-1}\mathbf{AX}$.

We now only need to assign the initial state set. We base our calculation of the initial state set on the final state sequence. This is done to ensure that the initial state set generates the final state sequence $f = f_0 f_1 \dots f_m$ and not any other sequence. With this in mind we examine the normal form characteristic matrix $\mathbf{A}$ of M'.

As we discussed in section 5.1.1, if the active state set is encoded as the row vector $\mathbf{y}$ then $\mathbf{A}\mathbf{y}^T$ will be the row vector representation of the next active state set. We now construct a row vector representation of $\mathbf{y}(0) = [y_0(0), y_1(0), \dots, y_{n'-1}(0)]$ and define the row vectors representing the active state sets as $\mathbf{y}(t) = [y_0(t), y_1(t), \dots, y_{n'-1}(t)] = \mathbf{A}\mathbf{y}^T(t-1) = \mathbf{A^2}\mathbf{y}^T(t-2) = \dots = \mathbf{A}^t\mathbf{y}^T(0)$.

As the final state sequence is generated by the active state sets containing final states, it follows that $f_i = 1$ if and only if $y_{n'-1}(i) = 1$, so that

$$f_i \;=\; y_{n'-1}(i). \tag{5.1}$$

From this we have that

$$y_{n'-1}(0) \;=\; f_0.$$

This means that the state $q_{n'-1}$ is an initial state if and only if the first bit in the final state sequence is a 1. Furthermore, as $\mathbf{y}(t) = \mathbf{A}\mathbf{y}^T(t-1)$, we have

$$\Big[y_0(t), y_1(t), \dots, y_{n'-1(t)}\Big] \begin{bmatrix} 0 & 0 & \dots & 0 & c_0 \\ 1 & 0 & \dots & 0 & c_1 \\ 0 & 1 & \dots & 0 & c_2 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & c_{n'-2} \\ 0 & 0 & \dots & 1 & c_{n'-1} \end{bmatrix} \begin{bmatrix} y_0(t-1) \\ y_1(t-1) \\ y_2(t-1) \\ \vdots \\ y_{n'-2}(t-1) \\ y_{n'-1}(t-1) \end{bmatrix},$$

and hence

$$\begin{aligned} y_0(t) &= c_0 y_{n'-1}(t-1), \text{ and} \\ y_i(t) &= y_{i-1}(t-1) + c_i y_{n'-1}(t-1), \text{ for } 1 \le i < n'. \end{aligned} \tag{5.2}$$

It follows that the state q_0 is active at time stamp t if and only if $c_0 = 1$ and the state $q_{n'-1}$ was active at time stamp $t-1$. Similarly the state q_i is active at time stamp t if either $c_i = 1$ and the state $q_{n'-1}$ was active at time stamp $t-1$ or the state q_{i-1} was active at time stamp $t-1$, but not both at the same time.

Using these equations we obtain the following for the second output symbol:

$$\begin{aligned} f_1 &= y_{n'-1}(1) \text{ from equation 5.1} \\ &= y_{n'-2}(0) + c_{n'-1} y_{n'-1}(0) \text{ from equation 5.2} \\ &= y_{n'-2}(0) + c_{n'-1} f_0 \text{ from equation 5.1,} \end{aligned}$$

so that

$$y_{n'-2}(0) = f_1 + c_{n'-1} f_0.$$

Continuing with this train of thought for the output symbols we have:

$$\begin{aligned}
f_2 &= y_{n'-1}(2) \\
&= y_{n'-2}(1) + c_{n'-1} y_{n'-1}(1) \\
&= y_{n'-2}(1) + c_{n'-1} f_1 \\
&= y_{n'-3}(0) + c_{n'-2} y_{n'-1}(0) + c_{n'-1} f_1 \\
&= y_{n'-3}(0) + c_{n'-2} f_0 + c_{n'-1} f_1, \\
\vdots &\quad \vdots \\
f_{n'-1} &= y_{n'-1}(n'-1), \\
&= y_0(0) + \ldots + c_{n'-2} f_{n'-3} + c_{n'-1} f_{n'-2},
\end{aligned}$$

so that

$$\begin{aligned}
y_{n'-3}(0) &= f_2 + c_{n'-2} f_0 + c_{n'-1} f_1, \\
\vdots &\quad \vdots \\
y_0(0) &= f_{n'-1} + \ldots + c_{n'-2} f_{n'-3} + c_{n'-1} f_{n'-2}.
\end{aligned}$$

The initial state set can be calculated using the vector $\mathbf{y}(0)$ as $\mathbf{y}(0)$ represents the initial active state set and thus the initial state set. Thus the state $q'_i \in Q'$ is an initial state if and only if $y_i(0) = 1$ as calculated above.

Examining the running example we have:

Example 5.12 (Reconstruction)

Continuing from example 5.10 (page 81) we have the final state sequence $f = 00110$ *with cycle length* 5 *and characteristic polynomial* $c'(X) = 1 + X^1 + X^2 + X^3 + X^4$. *Using* f *and* $c'(X)$ *we can construct the unary* $\oplus$*-FA* $M' = (Q', \{a\}, \delta', Q'_0, F', \oplus)$ *as follows:*

- *The state set* Q' *is simply* $Q' = \{q'_0, q'_1, q'_2, q'_3\}$.
- *The characteristic matrix* $\mathbf{A}'$ *of* M' *is:*

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

so that the transition function δ' *is given by (see also figure 5.2)*

δ'	a
q'_0	$\{q'_1\}$
q'_1	$\{q'_2\}$
q'_2	$\{q'_3\}$
q'_3	$\{q'_0, q'_1, q'_2, q'_3\}$

.

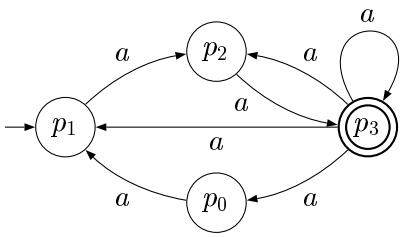

Figure 5.2: Example 5.12

- *The final state set F' is chosen as $F' = \{q'_3\}$.*

- *Lastly we calculate the initial state set as follows:*

$$\begin{aligned} y_0(3) &= f_0 = 0 \\ y_0(2) &= f_1 + c_3 f_0 = 0 + 0 = 0 \\ y_0(1) &= f_2 + c_2 f_0 + c_3 f_1 = 1 + 0 + 0 = 1 \\ y_0(0) &= f_3 + c_1 f_0 + c_2 f_1 + c_3 f_2 = 1 + 1 + 0 + 0 = 0, \end{aligned}$$

so that $Q'_0 = \{q'_1\}$.

Examining the resultant unary $\oplus$-FA we have that the state sequence is given by

$$\{q'_1\},\ \{q'_2\},\ \{q'_3\},\ \{q'_0, q'_1, q'_2, q'_3\},\ \{q'_0\},\ \{q'_1\},$$

which results in the final state sequence of 00110 *with cycle length* 5 *which is identical to the original final state sequence f.*

This completes the unary $\oplus$-FA minimization algorithm. We can now use these three parts of the minimization algorithm to formally define the minimization algorithm as follows:

Algorithm 5.13 (Unary $\oplus$-FA Minimization Algorithm)
Given a unary $\oplus$-FA M, the algorithm outputs the minimal $\oplus$-FA M' with one final state.

1. *Find the final state sequence f associated with M.*

2. *Apply the modified Berlekamp-Massey algorithm in algorithm 5.9 (page 80), given f to produce a polynomial $c(X)$ with the least degree such that $c(X)$ is the characteristic polynomial of a LFSR producing f.*

3. *Use $c(X)$ to construct a minimal $\oplus$-FA M' with one final state as described in section 5.1.3, such that $\mathcal{L}(M) = \mathcal{L}(M')$.*

Theorem 5.14 *Let M be a n-state unary $\oplus$-FA which accepts the language L. If we apply algorithm 5.13 to M, the resultant output is a minimal m-state unary $\oplus$-FA M' with one final state and $m \leq n$ which recognizes the language L.*

Proof Let M be a n-state unary $\oplus$-FA which accepts the language L.

Step 1 The first step of the algorithm consists of the construction of the final state sequence of M; that is, it is a binary encoding of the (unary) words accepted by M. The validity of the construction follows from the subset construction [15, 18].

Step 2 In the second step, the LFSR equivalent to M is constructed. The validity of this step follows from the equivalence between unary $\oplus$-FAs and LFSRs shown in [18]. The Berlekamp-Massey algorithm is guaranteed to result in a minimal LFSR with the same final state sequence as the original LFSR [1]. It follows that the LFSR resulting from the application of the algorithm represents a smallest LFSR which produces the same final state sequence, and hence that the corresponding $\oplus$-FA is a minimal $\oplus$-FA with the required sequence of final states.

Step 3 It is shown in [18] that for any j-state LFSR, there is a j-state unary $\oplus$-FA with the same cycle structure. In particular, from [5] the normal form given as the matrix $\mathbf{A}$, has the required cycle structure. Therefore, the transition function as given for M' is valid.

It remains to show that the constructed $\oplus$-FA accepts the same language as the original $\oplus$-FA. This follows trivially from the fact that in Step 2 above, the two LFSRs produce the same final state sequence. Hence, with the choice of initial states as derived from the final state sequence, and the fact that the final state sequences are identical, the assumption holds.

□

We calculate the computational complexity of the algorithm on the calculations involved in the three steps. In the first step the final state sequence has to be constructed. To this end on-the-fly subset construction is used which has worst case computational complexity of $\mathcal{O}(2^n)$ for a FA of n states. As each state set is to be recorded to identify the cycle, the space complexity of this step is also $\mathcal{O}(2^n)$.

The second step consists of the modified Berlekamp-Massey algorithm. The original algorithm has computational complexity of $\mathcal{O}(m^2)$ [12] where m is the length of the input sequence so that $m \leq 2^n$. Our modification requires the cycle of the sequence to be repeated until the characteristic polynomial is unmodified for the length of the sequence. The degree of the

characteristic polynomial is limited by the length of the sequence [7]. As algorithm 5.9 (page 80) increases the degree of the sequence according to a sub step of Step $2(b)$, the number of possible iterations of the algorithm is limited by the formula $L = t + 1 - L$ and the length of the sequence. The modified algorithm can thus repeat the original algorithm at most m times so that the modified algorithm has computational complexity of $\mathcal{O}(m^3)$.

The final step consists of the reconstruction of the unary $\oplus$-FA from the final state sequence and characteristic polynomial. Using the normal form of the characteristic matrix the calculation and creation of the state set and transition function is linear. As the state set has at most m states the complexity of this step is $\mathcal{O}(m)$.

The creation of the final state set and the alphabet has computational complexity of $\mathcal{O}(1)$.

Lastly the creation of the initial state set requires i operations per state $q'_i \in Q'$. As there are at most m states this step takes at most $0+1+\ldots+m-1$ operations and so has worst case computational complexity of $\mathcal{O}(m^2)$.

The algorithm therefore is capped at the first step with a computational complexity of $\mathcal{O}(2^n)$ where n is the number of states of the original $\oplus$-FA. However, if the final state sequence is given without computation the algorithm has complexity $\mathcal{O}(m^3)$ where m is the length of the input sequence.

5.1.4 Extensions

We have presented an algorithm which produces a minimal unary $\oplus$-FA with one final state. We now present some more examples of this minimization procedure as well as discussing the problems in extending the algorithm to handle more than one final state as well as generalizing the algorithm to non-unary languages.

We begin by providing an example where the final state sequence has a preamble, that is, where the cycle length is smaller than the sequence length. After that, we provide an example where the sequence contains a cycle of 0s so that words of length greater than a certain length are not accepted. Such sequences represent finite unary languages.

Example 5.15 (Unary $\oplus$-FA with Preamble)
Let $M = (\{q_0, q_1, q_2, q_3\}, \{a\}, \delta, \{q_0\}, \{q_3\}, \oplus)$ *with* δ *given by (see also*

[7] It is easy to verify that a final state sequence of length n can be generated by a unary $\oplus$-FA with n states. This implies that the degree of the characteristic polynomial generated by the modified Berlekamp-Massey algorithm must be smaller or equal to the length of the final state sequence.

figure 5.3, left)

δ	a
q_0	$\{q_1, q_2\}$
q_1	$\{q_2, q_3\}$
q_2	$\{q_1, q_2\}$
q_3	$\{q_0, q_1\}$

.

The state sequence is then given by:

$$\{q_0\},\ \{q_1, q_2\},\ \{q_1, q_3\},\ \{q_0, q_1, q_2, q_3\},\ \{q_0, q_1, q_2, q_3\}, \ldots$$

so that the final state sequence is given by

$$f \ = \ 0011,$$

with a cycle of length 1*. The preamble is thus* 001*.*

Note that this sequence can actually be written as 001 *with preamble* 00 *and cycle of length* 1*, although inputting either sequence into the modified Berlekamp-Massey algorithm will output the same result.*

The modified Berlekamp-Massey algorithm runs as follows:

s	d	$T(X)$	$c(X)$	L	m	$B(X)$	N
$-$	$-$	0	1	0	-1	1	0
0	0	0	1	0	-1	1	1
0	0	0	1	0	-1	1	2
1	1	1	$1+X^3$	3	2	1	3
1	1	$1+X^3$	$1+X^2+X^3$	3	2	1	4
1	0	$1+X^3$	$1+X^2+X^3$	3	2	1	5
1	1	$1+X^2+X^3$	$0+X^2+X^3$	3	2	1	6
1	1	$1+X^2+X^3$	$0+X^2+X^3$	3	2	1	7

so that the characteristic polynomial is $c(X) = X^2 + X^3$*.*

We can begin constructing the minimal unary $\oplus$*-FA* M' *with one final state as follows:*

$c(X)$ *has degree* 3 *so that* $M' = (\{q'_0, q'_1, q'_2\}, \{a\}, \delta', Q'_0, \{q'_2\}, \oplus)$ *.*

From $c(X)$ *we construct the normal form for the characteristic matrix*

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \quad ,$$

from which δ' *can be constructed as follows (see also figure 5.3, right):*

δ'	a
q'_0	$\{q'_1\}$
q'_1	$\{q'_2\}$
q'_2	$\{q'_2\}$

.

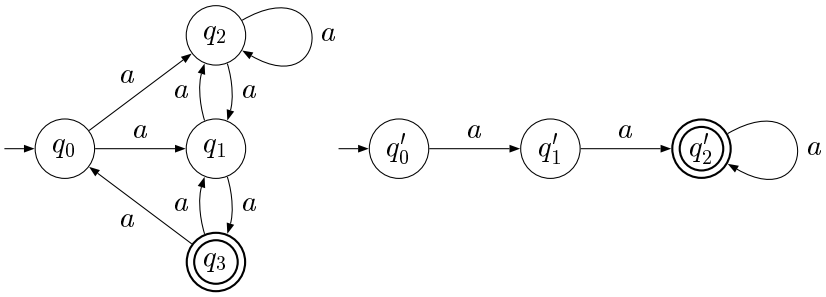

Figure 5.3: Example 5.15: Left M; Right M'

From $c(X)$ and f we determine the set of initial states as follows:

$$\begin{aligned} y_0(2) &= f_0 = 0, \\ y_0(1) &= f_1 + c_2 f_0 = 0 + 0 = 0, \text{ and} \\ y_0(0) &= f_2 + c_1 f_0 + c_2 f_0 = 1 + 0 + 0 = 1 \ , \end{aligned}$$

so that $Q_0' = \{q_0'\}$.
The resultant unary $\oplus$-FA produces the state sequence

$$\{q_0'\},\ \{q_1'\},\ \{q_2'\},\ \{q_2'\},$$

which produces the final state sequence 001 *with cycle length of* 1 *which is equivalent to the original final state sequence.*

It is of interest to note that the resultant unary $\oplus$-FA is in fact the minimal DFA.

Final state sequences with preambles usually result in some states being formed which are only active during the preamble of the sequence. Such states have the property that there are no transitions from the final state to it or any states which occur before it. In example 5.15 q_1' is such a state as the only state occurring previous to it is q_0' and there are no transitions from q_2' to q_0' or q_1'.

Example 5.16 (Unary $\oplus$-FA with Zero Cycle)

Let $M = (\{q_0, q_1, q_2, q_3\}, \{a\}, \delta, \{q_0\}, \{q_3\}, \oplus)$ with δ given by (see also figure 5.4, left)

δ	a
q_0	$\{q_0, q_3\}$
q_1	$\{q_0, q_2\}$
q_2	$\{q_0, q_1\}$
q_3	$\{q_0, q_2\}$

.

The state sequence is then given by:

$$\{q_0\},\ \{q_0, q_3\},\ \{q_2, q_3\},\ \{q_1, q_2\},\ \{q_1, q_2\},$$

so that the final state sequence is given by

$$f \;=\; 0110,$$

with a cycle of length 1. *The preamble is thus* 011.
The modified Berlekamp-Massey algorithm runs as follows:

s	d	$T(X)$	$c(X)$	L	m	$B(X)$	N
$-$	$-$	0	1	0	-1	1	0
0	0	0	1	0	-1	1	1
1	1	1	$1+X^2$	2	1	1	2
1	1	$1+X^2$	$1+X^1+X^2$	2	1	1	3
0	0	$1+X^2$	$1+X^1+X^2$	2	1	1	4
0	1	$1+X^1+X^2$	$1+X^1+X^2+X^3$	3	4	$1+X^1+X^2$	5
0	1	$1+X^1+X^2+X^3$	$0+X^3$	3	4	$1+X^1+X^2$	6
0	0	$1+X^1+X^2+X^3$	$0+X^3$	3	4	$1+X^1+X^2$	7 ,

so that the characteristic polynomial is $c(X) = X^3$.
We can begin constructing the minimal unary $\oplus$*-FA,* M' *with one final state as follows:*
$c(X)$ *has degree* 3 *so that* $M' = (\{q'_0, q'_1, q'_2\}, \{a\}, \delta', Q'_0, \{q'_2\}, \oplus)$.
From $c(X)$ *we construct the normal form for the characteristic matrix*

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad ,$$

from which δ' *can be constructed as follows (see also figure 5.4, right):*

δ'	a
q'_0	$\{q'_1\}$
q'_1	$\{q'_2\}$
q'_2	$\emptyset$

.

From $c(X)$ *and* f *we determine the set of initial states as follows:*

$$\begin{aligned} y_0(2) &= f_0 = 0, \\ y_0(1) &= f_1 + c_2 f_0 = 1 + 0 = 1, \ \textit{and} \\ y_0(0) &= f_2 + c_1 f_0 + c_2 f_0 = 1 + 0.1 + 0 = 1 \ , \end{aligned}$$

so that $Q'_0 = \{q'_0, q'_1\}$.
The resultant unary $\oplus$*-FA produces the following state sequence*

$$\{q'_0, q'_1\},\ \{q'_1, q'_2\},\ \{q'_2\},\ \emptyset,\ \emptyset,$$

which produces the final state sequence 0110 *with cycle length of* 1 *which is equivalent to the original final state sequence.*

We have taken a look at three possible sequence types which are possible in the final state sequence, that being

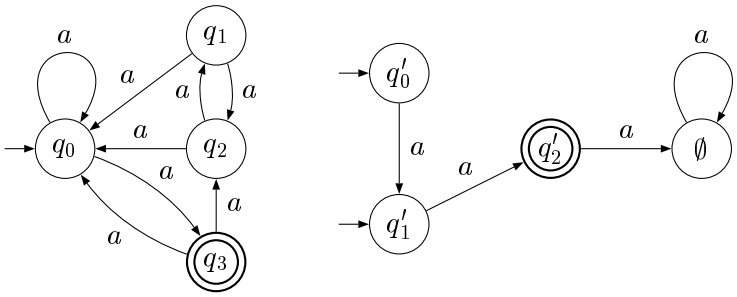

Figure 5.4: Example 5.16: Left M; Right M'

- full cycle,
- incomplete cycle or preamble, and
- cycle consisting only of 0s[8].

These three cases provide different types of unary $\oplus$-FAs with one final state. Sequences with a full cycle result in unary $\oplus$-FAs where the initial state set will be reached after a certain amount of character inputs. Sequences with the incomplete cycle either creates states for the purpose of the preamble, where these states are never in the active state set after the preamble, or the initial state set of the unary $\oplus$-FA generates a cycle in which the initial state set does not feature again. Sequences whose cycles consist only of zeros creates a unary $\oplus$-FA in which the null state will be generated after a certain number of iterations. In this case there are either no transitions from the final state, or the transitions from the final state interact with other transitions resulting in an empty active state set.

We stated previously that the unary $\oplus$-FA generated from this algorithm is not always minimal with respect to unary $\oplus$-FAs with more than one final state. In the next example we examine a situation where this is the case, after which we discuss the possibility of extending the algorithm to more than one final state.

Example 5.17 (Minimal with Multiple Final States)

This example is divided into two parts. In the first part we attempt to minimize a unary $\oplus$-FA to a unary $\oplus$-FA with one final state. Although the resultant unary $\oplus$-FA will be minimal among unary $\oplus$-FA with one final state it will be easy to verify that it is in fact not minimal among all unary $\oplus$-FA. In the second part of the example we show how a minimal unary $\oplus$-FA with more than one final state can be constructed.

[8]Representing languages of finite length.

Let $M = (\{q_0, q_1, q_2, q_3\}, \{a\}, \delta, \{q_0\}, \{q_1, q_3\}, \oplus)$ *with* δ *given by (see also figure 5.5, page 94)*

δ	a
q_0	$\{q_1\}$
q_1	$\{q_3\}$
q_2	$\{q_1\}$
q_3	$\{q_2, q_3\}$

.

The state sequence is then given by:

$$\{q_0\},\ \{q_1\},\ \{q_3\},\ \{q_2, q_3\},\ \{q_1, q_2, q_3\},\ \{q_1, q_2\},\ \{q_1, q_3\},\ \{q_2\},\ \{q_1\},\ \ldots$$

so that the final state sequence is given by

$$f\ =\ 01111110,$$

with a cycle of length 7*. The preamble is thus* 0*.*

Note that this sequence can actually be written as 0111111 *with no preamble and cycle of length* 7*, although inputting either sequence into the modified Berlekamp-Massey algorithm will output the same result.*

The modified Berlekamp-Massey algorithm runs as follows:

s	d	$T(X)$	$c(X)$	L	m	$B(X)$	N
$-$	$-$	0	1	0	-1	1	0
0	0	0	1	0	-1	1	1
1	1	1	$1+X^2$	2	1	1	2
1	1	$1+X^2$	$1+X^1+X^2$	2	1	1	3
1	1	$1+X^1+X^2$	$0+X^1+X^2$	2	1	1	4
1	0	$1+X^1+X^2$	$0+X^1+X^2$	2	1	1	5
1	0	$1+X^1+X^2$	$0+X^1+X^2$	2	1	1	6
1	0	$1+X^1+X^2$	$0+X^1+X^2$	2	1	1	7
0	1	$0+X^1+X^2$	$1+X^5+X^6$	6	7	$0+X^1+X^2$	8
1	0	$0+X^1+X^2$	$1+X^5+X^6$	6	7	$0+X^1+X^2$	9
1	1	$1+X^5+X^6$	$1+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	10
1	0	$1+X^5+X^6$	$1+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	11
1	1	$1+X^3+X^4+X^5+X^6$	$1+X^1+X^2+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	12
1	0	$1+X^3+X^4+X^5+X^6$	$1+X^1+X^2+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	13
1	0	$1+X^3+X^4+X^5+X^6$	$1+X^1+X^2+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	14
0	0	$1+X^3+X^4+X^5+X^6$	$1+X^1+X^2+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	15
1	0	$1+X^3+X^4+X^5+X^6$	$1+X^1+X^2+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	16
1	0	$1+X^3+X^4+X^5+X^6$	$1+X^1+X^2+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	17
1	0	$1+X^3+X^4+X^5+X^6$	$1+X^1+X^2+X^3+X^4+X^5+X^6$	6	7	$0+X^1+X^2$	18

so that the characteristic polynomial is $c(X) = 1+X^1+X^2+X^3+X^4+X^5+X^6$*.*

We can begin constructing the minimal unary $\oplus$*-FA,* M' *with one final state as follows:*

$c(X)$ *has degree* 6 *so that* $M' = (\{q'_0, q'_1, q'_2, q'_3, q'_4, q'_5\}, \{a\}, \delta', Q'_0, \{q'_5\}, \oplus)$*.*

From $c(X)$ we construct the normal form for the characteristic matrix

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} ,$$

from which δ' can be constructed as follows (see also figure 5.5, right):

δ'	a
q'_0	$\{q'_1\}$
q'_1	$\{q'_2\}$
q'_2	$\{q'_3\}$
q'_3	$\{q'_4\}$
q'_4	$\{q'_5\}$
q'_5	$\{q'_0, q'_1, q'_2, q'_3, q'_4, q'_5\}$

.

From $c(X)$ and f we determine the set of initial states as follows:

$$\begin{aligned}
y_0(5) &= f_0 = 0, \\
y_0(4) &= f_1 + c_5 f_0 = 1 + 0 = 1, \\
y_0(3) &= f_2 + c_4 f_0 + c_5 f_1 = 1 + 1.1 + 0 = 0, \\
y_0(2) &= f_3 + c_3 f_0 + c_4 f_1 + c_5 f_2 = 1 + 1.1 + 1.1 + 0 = 1, \\
y_0(1) &= f_4 + c_2 f_0 + c_3 f_1 + c_4 f_2 + c_5 f_3 = 1 + 1.1 + 1.1 + 1.1 + 0 = 0, \text{ and} \\
y_0(0) &= f_5 + c_1 f_0 + c_2 f_1 + c_3 f_2 + c_4 f_3 + c_5 f_4 = 1 + 1.1 + 1.1 + 1.1 + 1.1 + 0 = 1,
\end{aligned}$$

so that $Q'_0 = \{q'_0, q'_2, q'_4\}$.

The resultant unary $\oplus$-FA produces the state sequence

$$\{q'_0, q'_2, q'_4\},\ \{q'_1, q'_3, q'_5\},\ \{q'_0, q'_1, q'_3, q'_5\},\ \{q'_0, q'_3, q'_5\},\ \{q'_0, q'_2, q'_3, q'_5\},$$
$$\{q'_0, q'_2, q'_5\},\ \{q'_0, q'_2, q'_4, q'_5\},\ \{q'_0, q'_2, q'_4\},\ \ldots ,$$

which produces the final state sequence 0111111 *with cycle length of* 7 *which is equivalent to the original final state sequence.*

It is easy to verify that although M' is the minimal unary $\oplus$-FA with one final state, it is not a minimal unary $\oplus$-FA, as the original $\oplus$-FA has four states while M' has six states.

To construct a minimal unary $\oplus$-FA with more than one final state we need to first choose the number of final states, and assign new final state sequences to each final state.

In our case we will deal with a unary $\oplus$-FA with exactly two final states. We need to determine two final state sequences whose union equals the original final state sequence. Through trial and error we obtain the two final state sequences, f' and f'', as:

$$f' = 0100111, \text{ and } f'' = 0011101,$$

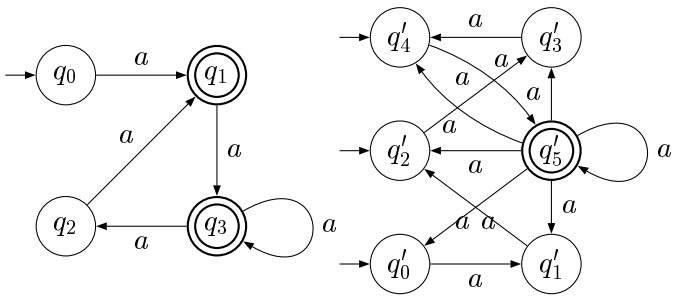

Figure 5.5: Example 5.17: Left M; Right M'

both with cycles of length 7.

We then apply the modified Berlekamp-Massey algorithm on both f_1 *and* f_2 *resulting in two characteristic polynomials* $c_1(X)$ *and* $c_2(X)$ *respectively.*

Running the algorithm we have that $c_1(X) = 1 + X^2 + X^3 = c_2(X)$*. As both sequences have identical characteristic polynomials we can attempt to construct a merged unary* $\oplus$*-FA* $M'' = (\{q''_0, q''_1, q''_2\}, \{a\}, \delta'', Q''_0, F'', \oplus)$ *using the two sequences. This is done by first constructing* δ'' *from the normal form characteristic matrix*

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} ,$$

so that δ'' *can constructed be as follows (see also figure 5.6):*

δ''	a
q''_0	$\{q''_1\}$
q''_1	$\{q''_2\}$
q''_2	$\{q''_0, q''_2\}$

.

Next we need to assign two states to be final states. We choose the first as the last state as in the previous examples, that is $q''_2 \in F''$*. The second we choose arbitrarily as* $q''_0 \in F''$*, so that* $F'' = \{q''_0, q''_2\}$*. We need to assign each final state to a final state sequence, and in this case we assign* q''_0 *to* f'' *and* q''_2 *to* f'*.*

To calculate the initial state set we need to examine both final states with their respective final state sequences. This is done as follows:

	f'_i	$=$	$y_2(i)$	f''_i	$=$	$y_0(i)$	
	f'_0	$=$	$y_2(0)$	f''_0	$=$	$y_0(0)$	
so that	0	$=$	$y_2(0)$	0	$=$	$y_0(0)$	
Next	f'_1	$=$	$y_2(1)$	f''_1	$=$	$y_0(1)$	
	f'_1	$=$	$y_2(0) + y_1(0)$	f''_1	$=$	$y_2(0)$	from equation 5.2
so that	1	$=$	$0 + y_1(0)$	0	$=$	$y_2(0)$	
and	1	$=$	$y_1(0)$	0	$=$	$y_2(0)$	,

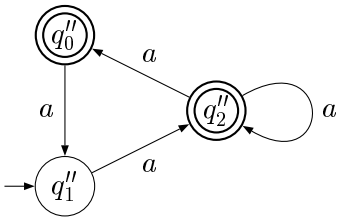

Figure 5.6: Example 5.17: M''

so that $Q_0'' = \{q_1''\}$.
The resultant unary $\oplus$*-FA produces the following state sequence*

$$\{q_1''\},\ \{q_2''\},\ \{q_0'', q_2''\},\ \{q_0'', q_1'', q_2''\},\ \{q_0'', q_1''\},\ \{q_1'', q_2''\},\ \{q_0''\},\ \{q_1''\},\ \cdots$$

which produces the final state sequence 0111111 *with cycle length of* 7 *which is equivalent to the original final state sequence.*

From the second part of the example we see that it is possible to combine two or more unary $\oplus$-FA with one final state to create a unary $\oplus$-FA with multiple final states. However, the computational complexity of generalizing this becomes impractical. This is because each 1 in the final state sequence needs to be assigned to whichever final states are active at the time. For example, for two final states, each 1 in the final state sequence must be assigned to the first, the second or to both final states. This means that for m final states and k 1's there are $(2^m - 1)^k$ possible final state assignments. These assignments include all assignments to less than m final states.

So that, although it is possible to extend this algorithm to include multiple final states, the computational complexity cost is too high for a practical algorithm.

In the non-unary case, the problem with extending this algorithm is that non unary $\oplus$-FAs is not related to LFSRs. Some other issues facing the generalization of this algorithm include that the characteristic matrix is replaced with a series of characteristic matrices for each alphabet symbol, and as such a characteristic polynomial is not as easily obtainable from these characteristic matrices. Similarly one is not faced with a single state or final state sequence but instead with $|\Sigma|^n$ possible sequences of length n. This means that any algorithm would need to handle each possible sequence and the problem becomes NP-Hard.

For these reasons we examine other ways to minimize or reduce the number of states in $\oplus$-FA. We begin by discussing a modification of Kameda and Weiner's algorithm to minimize NFAs.

5.2 General $\oplus$-FAs

In the previous section we discussed the minimization of a simplified case of the $\oplus$-FAs, that of the unary $\oplus$-FA. We developed an algorithm which works to produce the unary minimal $\oplus$-FA with one final state and discussed how this algorithm could be extended to the general case. However, this extension increased the algorithms computational complexity and caused the algorithm to become impractical.

In this section we look at reduction and minimization algorithms used for conventional FAs and attempt to adapt them to the general case of $\oplus$-FAs.

The minimization and reduction algorithms for FAs in chapter 4 were based on the detection and removal of reducible states. We identified five types of reducible states for NFAs from definitions 4.4 and 4.22 (page 31), namely *initial-useless*, *final-useless*, *redundant*, *equivalent* and *contained*. However, not all of these reducible states are applicable to $\oplus$-FAs.

We begin by discussing reducible states of $\oplus$-FAs, followed by the three categories of FA minimization and their relationship to $\oplus$-FAs. These are *language construction*, *transformation construction* based on Kameda and Weiner's NFA minimization algorithm [11] and *state comparison* which borrows from the state reduction algorithms by Ilie and Yu [9].

5.2.1 Reducible States

In order to construct minimization and reduction algorithms we need to be able to distinguish which, if any, states in a $\oplus$-FA can be reduced. We begin by examining the reducible states as defined for FAs in definitions 4.4 and 4.22 (pages 31 and 54). These definitions are based on the predecessors and successors[9] of the states of the $\oplus$-FA.

We examine five types of the reducible states, these being initial-useless, final-useless, redundant, contained and equivalent. In each case, we characterize the reducible state, provide an example of the reducible state, and then formally prove whether these states are in fact reducible in a $\oplus$-FA.

Initial-Useless States

An *initial-useless* state is defined as a state in a FA which has an empty predecessor, that is, in a $*$-FA $M = (Q, \Sigma, \delta, Q_0, F, *)$, a state q is *initial-useless* if

$$pr(q) \;=\; \emptyset.$$

[9] See definition 4.2, page 30.

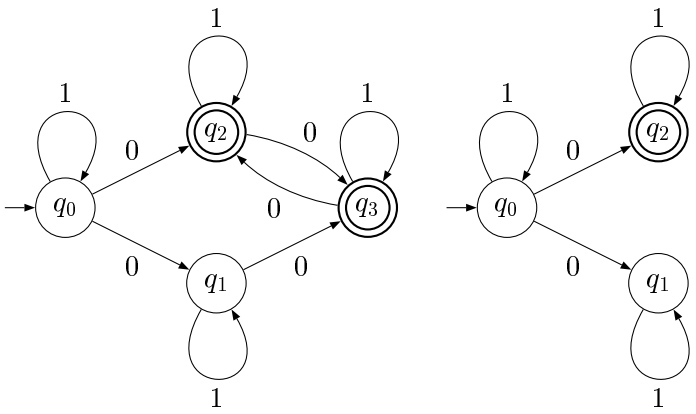

Figure 5.7: Example 5.18: Left M; Right M'

Example 5.18 (Initial-Useless States)

Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_0\}, \{q_2, q_3\}, \oplus)$, where δ is given by (see also figure 5.7)

δ	0	1
q_0	$\{q_1, q_2\}$	$\{q_0\}$
q_1	$\{q_3\}$	$\{q_1\}$
q_2	$\{q_3\}$	$\{q_2\}$
q_3	$\{q_2\}$	$\{q_3\}$

.

We begin by examining the transition function of the DFA equivalent to M:

	0	1
$[q_0]$	$[q_1, q_2]$	$[q_0]$
$[q_1, q_2]$	$\emptyset$	$[q_1, q_2]$

.

As can be seen in the DFA's transition function, the state q_3 is unreachable, so that $pr(q_3) = \emptyset$.

By removing the state q_3 and all transitions to and from the state, we obtain the $\oplus$-FA $M' = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta', \{q_0\}, \{q_2\}, \oplus)$ where δ' is given by (see also figure 5.7)

δ	0	1
q_0	$\{q_1, q_2\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_1\}$
q_2	$\emptyset$	$\{q_2\}$

.

We have that $\mathcal{L}(M') = \mathcal{L}(M) = 1^*01^*$.

In example 5.18 above, state q_3 was identified as initial-useless and removed. However, the reader may note that if M were a traditional NFA, q_3 would not be initial-useless. This follows because there exists a transition from q_0 to q_1 on the symbol 0, and from q_1 to q_3 on symbol 0.

In a traditional NFA a state is initial-useless if it is not an initial state and there exists no transitions to it from any other non initial-useless state [11]. This follows from the fact that no series of transitions exists from an initial state to an initial-useless state. Thus, if the only transitions to a state are from initial-useless states, the state is initial-useless as all series of transitions which end on this state must pass through initial-useless states.

In a $\oplus$-FA however, a state is initial-useless if it is not an initial state and there exists no transitions to it from any active state set. Or, in other words, if that state does not occur in any state in the DFA.

Theorem 5.19 *Initial-useless states are reducible in a $\oplus$-FA.*

Proof In a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$, if a state $q \in Q$ is initial-useless, then for every word $y \in \Sigma^*$

$$q \notin \delta(Q_0, y).$$

To show that the initial-useless states are reducible we will construct a $\oplus$-FA where these states are removed and show that the two $\oplus$-FAs are equivalent, that is they accept the same language.

We generate the $\oplus$-FA $M' = (Q - \{q\}, \Sigma, \delta', Q_0, F - \{q\}, \oplus)$ with δ' described as

$$\delta'(q_i, \sigma) = \delta(q_i, \sigma) - \{q\},$$

where $q_i \in Q - \{q\}$, $\sigma \in \Sigma$ and q is initial-useless.

We begin by showing that for every $x \in \Sigma^*$

$$\delta(Q_0, x) \cap F = \delta'(Q_0, x) \cap F - \{q\},$$

after which we show that $\delta(Q_0, x) = \delta'(Q_0, x)$.

As q is initial-useless we have that $q \notin \delta(Q_0, x)$ where $x \in \Sigma^*$. This means that $\delta(Q_0, x) \cap F = \delta'(Q_0, x) \cap F - \{q\}$. This completes our first step, and next we show that $\delta(Q_0, x) = \delta'(Q_0, x)$.

For every $x \in \Sigma^*$ with $x = x_0 x_1 \dots x_m$ with $x_i \in \Sigma$ we have that

$$\begin{aligned} \delta'(Q_0, x_0 x_1 \dots x_m) &= \delta'(\dots \delta'(\delta'(Q_0, x_0), x_1) \dots, x_m) \\ &= \delta(\dots \delta(\delta(Q_0, x_0) - \{q\}, x_1) - \{q\}, \dots, x_m) - \{q\}. \end{aligned}$$

As q is initial-useless in M, we have that $q \notin Q_0$ and $q \notin \delta(Q_0, y)$ where $y \in \Sigma^*$. This means that with $i \leq n$, $q \notin \delta(Q_0, x_0 x_1 \dots x_i)$ so that

$$\begin{aligned} \delta'(Q_0, x_0) &= \delta(Q_0, x_0) - \{q\} \\ &= \delta(Q_0, x_0) \\ \delta'(Q_0, x_0 x_1) &= \delta(\delta(Q_0, x_0), x_1) - \{q\} \\ &= \delta(\delta(Q_0, x_0), x_1) \\ \vdots &= \vdots \\ \delta'(Q_0, x) &= \delta(\dots \delta(\delta(Q_0, x_0), x_1) \dots, x_m) - \{q\} \\ &= \delta(Q_0, x). \end{aligned}$$

We now have that $\delta(Q_0', x) = \delta(Q_0, x)$ and $\delta(Q_0, x) = \delta(Q_0, x) - \{q\}$ so that $\delta(Q_0', x) = \delta(Q_0, x) - \{q\}$. Thus if $x \in \mathcal{L}(M)$ then

$$\delta(Q_0, x) \cap F \;=\; \delta(Q_0, x) \cap F - \{q\} \;=\; \delta'(Q_0, x) \cap F - \{q\}$$

so that $x \in \mathcal{L}(M')$.

Similarly if $x \in \mathcal{L}(M')$ then

$$\delta'(Q_0, x) \cap F - \{q\} \;=\; \delta(Q_0, x) \cap F - \{q\} \;=\; \delta(Q_0, x) \cap F$$

so that $x \in \mathcal{L}(M)$.

Thus $\mathcal{L}(M) = \mathcal{L}(M')$. □

The theorem not only shows that all initial-useless states are reducible, but also presents an algorithm to remove them. That is, initial-useless states are reduced by removing them from the state set, the final state set and by removing all of their incoming and outgoing transitions.

The next problem is how to determine all of the initial-useless states. All possible active state sets have to be examined to be able to determine if a state is initial-useless. This can be done by using the DFA generated by the subset construction as follows:

Algorithm 5.20 (Initial-Useless States)

Let M be a $\oplus$-FA with $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$. The algorithm outputs the set I of all initial-useless states.

1. *Construct*[10] $M' \leftarrow \mathrm{DFA}(M) = (P, \Sigma, \delta', p_0, F)$.
2. *Construct* $I \leftarrow Q$.
3. *For every state* $p \in P$ *do*

   For every state $q \in p$, $I \leftarrow I - \{q\}$.
4. *Return* I.

Algorithm 5.20 works by first creating the DFA equivalent to the given $\oplus$-FA using on the fly subset construction so that the DFA contains no initial-useless states. A set of all states of the $\oplus$-FA is created representing the possible initial-useless states. Each state in the DFA is then processed and the states it represents in the $\oplus$-FA are removed from the initial-useless set. The remaining states in the set, after all DFA states have been processed, cannot be reached from the initial state set and are thus initial-useless.

[10] It is important to note here that M' is constructed using on-the-fly subset construction and so contains no initial-useless states as described in convention 2.12, page 14.

Final-Useless States

Final-useless states are states whose successors are empty, that is in a $*$-FA $M = (Q, \Sigma, \delta, Q_0, F, *)$ a state $q \in Q$ is final-useless if for every $x \in \Sigma^*$

$$\delta(q, x) \ \cap \ F \ = \ \emptyset.$$

A state is final-useless in an NFA if there is no series of transitions from that state to any final state. However, in a $\oplus$-FA, a state may be final-useless although a series of transitions results in a final state. This is because transitions on the same alphabet symbol may result in the final state becoming inactive.

Example 5.21

Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, \{q_0\}, \{q_4\}, \oplus)$ with δ given by (see also figure 5.8):

δ	0	1
q_0	$\{q_1, q_2\}$	$\{q_2\}$
q_1	$\{q_2\}$	$\{q_4\}$
q_2	$\{q_1, q_3\}$	$\emptyset$
q_3	$\emptyset$	$\{q_4\}$
q_4	$\{q_4\}$	$\{q_4\}$

,

and $\mathcal{L}(M) = 01(0|1)^*$.

We begin by examining the transition function of the DFA equivalent to M (see also figure 5.8):

			0	1
p_0	=	$[q_0]$	$[q_1, q_2]$	$[q_2]$
p_1	=	$[q_1, q_2]$	$[q_1, q_2, q_3]$	$[q_4]$
p_2	=	$[q_2]$	$[q_1, q_3]$	$\emptyset$
p_3	=	$[q_1, q_2, q_3]$	$[q_1, q_2, q_3]$	$\emptyset$
p_4	=	$[q_4]$	$[q_4]$	$[q_4]$
p_5	=	$[q_1, q_3]$	$[q_2]$	$\emptyset$

.

By examining the DFA(M) (shown center in figure 5.8) we see that there is no series of transitions from the state $p_2 = [q_2]$ which can lead to the only final state in the DFA $p_4 = [q_4]$. That is no word $x \in \Sigma^*$ exists so that $f(q_2, x) \cap F \neq \emptyset$. Thus the state q_2 is final-useless. As the state q_2 is final-useless we can reduce M by removing state q_2 from Q, Q_0 and F, and remove all incoming and outgoing transitions accordingly.

The $\oplus$-FA $M' = (\{q_0, q_1, q_3, q_4\}, \{0, 1\}, \delta', \{q_0\}, \{q_4\}, \oplus)$ is constructed

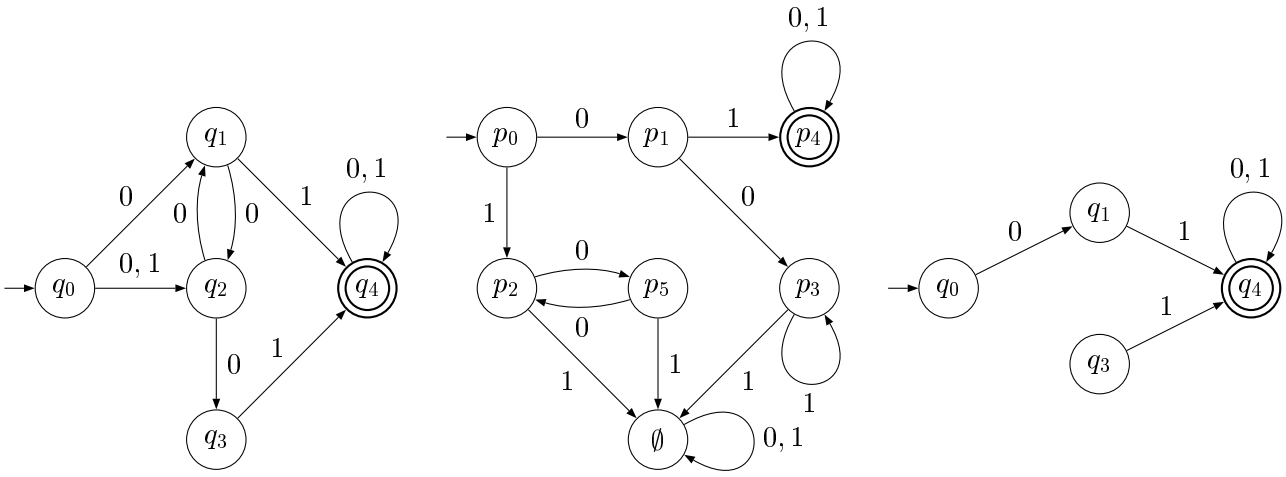

Figure 5.8: Example 5.21: Left M; Center DFA(M); Right M'

from M where δ' is given by (see also figure 5.8):

δ'	0	1
q_0	q_1	$\emptyset$
q_1	$\emptyset$	q_4
q_3	$\emptyset$	q_4
q_4	q_4	q_4

.

It is easy to verify that $\mathcal{L}(M') = \mathcal{L}(M) = 01(0|1)^*$. Furthermore, the state q_3 can then be seen as initial-useless and can be removed in M', resulting in a $\oplus$-FA with three states.

Before showing that final-useless states are reducible in $\oplus$-FAs we discuss some implications of the example. Firstly as shown at the end of example 5.21 above, the removal of final-useless states can generate initial-useless states. Similarly the removal of initial-useless states can generate other types of reducible states. This result holds for all types of reducible states that we shall identify and remove.

With the above in mind we can state the following theorem:

Theorem 5.22 *Final-useless states are reducible in a $\oplus$-FA.*

Proof In a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$, if a state $q \in Q$ is final-useless, then for every word $x \in \Sigma^*$

$$\delta(q, x) \cap F \;=\; \emptyset.$$

Therefore, for any $x = yz$ with $x, y, z \in \Sigma^*$ and $x \in \mathcal{L}(M)$ it follows that $\delta(Q_0, y) \neq q$ where q is final-useless.

Now generate the $\oplus$-FA $M' = (Q', \Sigma, \delta', Q_0', F, \oplus)$ where $Q' = Q - \{q\}$, $Q_0' = Q_0 - \{q\}$ with δ' described as

$$\delta'(q', \sigma) \;=\; \delta(q', \sigma) - \{q\},$$

where $q' \in Q'$ and $\sigma \in \Sigma$.

Note that we do not need to remove q from F as if $q \in F$ it cannot be final-useless.

For every $x \in \Sigma^*$ with $x = x_0 x_1 \dots x_m$ with $q \in \delta(Q_0, x_0 x_1 \dots x_{i-1})$ where $x_i \in \Sigma$ we have that

$$\begin{aligned} \delta'(Q_0, x_0 x_1 \dots x_m) &= \delta'(\dots \delta'(\delta'(Q_0, x_0), x_1) \dots, x_m) \\ &= \delta(\dots \delta(\delta(Q_0, x_0) - \{q\}, x_1) - \{q\} \dots, x_m) - \{q\} \end{aligned},$$

but as q is final-useless, we have that $\delta(q, x_i x_{i+1} \dots x_m) \cap F = \emptyset$ so that

$$\delta'(Q_0, x) \cap F \;=\; \delta(\dots \delta(\delta(Q_0, x_0), x_1) \dots, x_m) \cap F \;=\; \delta(Q_0, x) \cap F.$$

Thus as $\delta'(Q_0, x) \cap F \;=\; \delta(Q_0, x) \cap F$ we have that $\mathcal{L}(M) = \mathcal{L}(M')$. □

As with initial-useless states, to determine all of the final-useless states in a $\oplus$-FA can be done using the subset construction algorithm as follows:

Algorithm 5.23 (Final-Useless States)

Let M be a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$. The algorithm outputs the set I of all final-useless states.

1. *Construct*[11] $M' \leftarrow \mathrm{DFA}(M) = (2^Q, \Sigma, \delta', p_0, F')$ *constructed without removing the initial-useless states.*
2. *Construct* $I \leftarrow Q - F$.
3. *For every state* $p \in 2^Q$ *do*

   If $p = \{q\}$ *with* $q \in I$ *and* p *is not final-useless in* M', *then*

   $$I \leftarrow I - \{q\}.$$

4. *Return* I.

Algorithm 5.23 works by first creating the DFA equivalent to the given $\oplus$-FA not using on the fly subset construction so that the DFA contains initial-useless states. A set I of all states excluding final states of the $\oplus$-FA is created, which represents the possible final-useless states. Each state in the DFA which corresponds to exactly one state in the $\oplus$-FA is then processed. If this state is not final-useless, that is a series of transitions from it to a final state in the DFA can be found, then this state is removed from the possible final-useless set. After each applicable state has been processed in the DFA, the remaining states in set I are final-useless.

[11] It is important to note here that M' is not constructed using on-the-fly subset construction and so contains initial-useless states as described in definition 2.11, page 13.

Redundant States

Redundant states are states which have identical predecessors, that is in a $*$-FA $M = (Q, \Sigma, \delta, Q_0, F, *)$ a state $q_i \in Q$ is redundant with respect to a state $q_j \in Q$ if for every word $x \in \Sigma^*$,

$$q_i \in \delta(Q_0, x) \;\Leftrightarrow\; q_j \in \delta(Q_0, x).$$

As with the final-useless states, detection of redundant states in a $\oplus$-FA can be complicated by the interaction of transitions due to the $\oplus$-operator. Where in an NFA we could simply compare parent states, the combinations of parent states in a $\oplus$-FA complicate the matter due to the $\oplus$-operator.

We begin with an example of the reduction of redundant states in a $\oplus$-FA after which we show that all redundant states are reducible and how they can be detected.

Example 5.24

Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, \{q_0\}, \{q_4\}, \oplus)$ with δ given by (see also figure 5.9):

δ	0	1
q_0	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_2, q_3, q_4\}$	$\{q_0\}$
q_2	$\{q_2\}$	$\{q_0, q_1\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_1\}$

.

Examining M, we now show that state q_3 is redundant with respect to the state q_4. We note that the parents[12] *of q_3 and q_4 are given by $pa(q_3) = \{q_1\}$ and $pa(q_4) = \{q_1, q_3, q_4\}$. Now, we examine the complete set of transitions from the transition function of the DFA equivalent to M:*

	0	1
$[q_0]$	$[q_1]$	$[q_2]$
$[q_1]$	$[q_2, q_3, q_4]$	$[q_0]$
$[q_2]$	$[q_2]$	$[q_0, q_1]$
$[q_2, q_3, q_4]$	$[q_2]$	$[q_0]$
$[q_0, q_1]$	$[q_1, q_2, q_3, q_4]$	$[q_0, q_2]$
$[q_1, q_2, q_3, q_4]$	$[q_3, q_4]$	$\emptyset$
$[q_0, q_2]$	$[q_1, q_2]$	$[q_0, q_1, q_2]$
$[q_3, q_4]$	$\emptyset$	$[q_1]$
$[q_1, q_2]$	$[q_3, q_4]$	$[q_1]$
$[q_0, q_1, q_2]$	$[q_1, q_3, q_4]$	$[q_1, q_2]$
$[q_1, q_3, q_4]$	$[q_2, q_3, q_4]$	$[q_0, q_1]$

.

[12] See definition 2.3, page 8.

We note that, in the transition table of the DFA, the state q_3 can only be active when q_4 is active and vice versa. Hence q_3 and q_4 are redundant with respect to each other.

To reduce the $\oplus$-FA we attempt to merge the redundant states into one state which represents both the original states. Thus transition sequences which result in the two original states being set active should result in the new state to be set active.

We begin with the incoming transition of the new state. We cannot simply assign the union of the original two states incoming transitions to the new state. This would result in the state never being set active, due to the nature of the symmetric difference operator. Instead, we assign only one of the two states' incoming transitions to the new state. As both are always set active it is not important which original state is chosen.

For the outgoing transitions we need the new state's outgoing transitions to result in the same active state set as the original states. This is done by assigning the symmetric difference of the original outgoing transitions to the new state.

When choosing the state to remove, it is important to note that if exactly one of the states is final then we should remove the non-final state. This is due to the fact that we wish to maintain the language of the $\oplus$-FA. If we remove the final state then words which would be accepted when reaching the final state now only reach the non-final state[13] *and so would not be accepted. This is not an issue when either both states are final or non-final as the words will be accepted or rejected in either state.*

We merge the states q_3 and q_4 by removing the state q_3, as q_4 is a final state. The state new outgoing transitions of the state q_4, $\delta'(q_4, \sigma)$, would be equal to the symmetric difference of the outgoing transitions of the original two states. So that, $\delta'(q_4, \sigma) = \delta(q_3, \sigma) \oplus \delta(q_4, \sigma)$ which for $\sigma \in \Sigma$ is:

$$\begin{aligned}
\delta'(q_4, 0) &= \delta(q_3, 0) \oplus \delta(q_4, 0) \\
&= \{q_4\} \oplus \{q_4\} \\
&= \emptyset \\
\delta'(q_4, 1) &= \delta(q_3, 1) \oplus \delta(q_4, 1) \\
&= \emptyset \oplus \{q_1\} \\
&= \{q_1\}
\end{aligned}$$

This results in the $\oplus$-FA M' as follows:

The $\oplus$-FA $M' = (\{q_0, q_1, q_2, q_4\}, \{0, 1\}, \delta', \{q_0\}, \{q_4\}, \oplus)$ is constructed

[13] Excluding the influence of other final states.

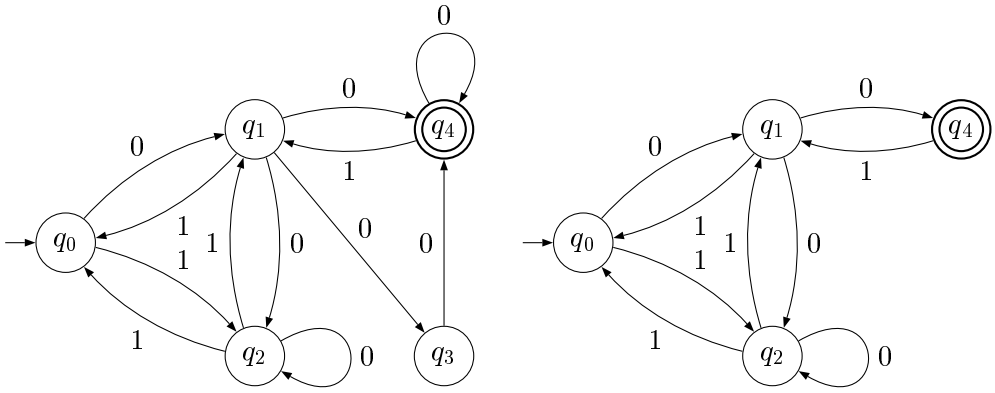

Figure 5.9: Example 5.24: Left M; Right M'

from M where δ' is given by (see also figure 5.9):

δ'	0	1
q_0	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_2, q_4\}$	$\{q_0\}$
q_2	$\{q_2\}$	$\{q_0, q_1\}$
q_4	$\emptyset$	$\{q_1\}$

.

It is easy to verify that $\mathcal{L}(M') = \mathcal{L}(M)$ by comparing the corresponding DFAs.

In example 5.24, the states q_3 and q_4 were redundant with respect to each other although they did not share parents. This was because the transitions from q_3 and q_4 interacted to not enable q_4 to be set active without q_3 being set active. The states q_3 and q_4 were merged by removing the state q_3 and merging the transitions from q_4 with the transitions of q_3 using the symmetric difference operator. As noted in the example the transitions to q_3 are not merged with those to q_4 as such an action would not allow the state q_4 to be set active.

Theorem 5.25 *Redundant states are reducible in a $\oplus$-FA.*

Proof In a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$, if the states $q_i, q_j \in Q$ are redundant with respect to each other, then for every word $x \in \Sigma^*$

$$q_i \in \delta(Q_0, x) \;\Leftrightarrow\; q_j \in \delta(Q_0, x).$$

We assume without loss of generality that either both $q_i, q_j \in F$ or $q_i \notin F$[14].

[14]We can assume this as either both states are final states, or at least one state is not a final state. In this second case we assign q_i to be a non-final state. Thus we only assign q_i to be a final state if both reducible states are final.

Now generate the $\oplus$-FA $M' = (Q', \Sigma, \delta', Q'_0, F', \oplus)$ where $Q' = Q - \{q_i\}$, $Q'_0 = Q_0 - \{q_i\}$, $F' = F - \{q_i\}$ where δ' is described as follows:

$$\delta'(q', \sigma) = \begin{cases} \delta(q', \sigma) - \{q_i\} & , \quad q' \neq q_j, \\ \delta(q', \sigma) \oplus \delta(q_i, \sigma) - \{q_i\} & , \quad q' = q_j. \end{cases}$$

If $q_i \in Q_0$, it follows that $q_j \in Q_0$. So that, with $\sigma \in \Sigma$ and $q_i \in Q_0$ we have that

$$\begin{aligned} \delta'(Q'_0, \sigma) &= \delta(Q_0 - \{q_i, q_j\}, \sigma) \oplus \delta'(q_j, \sigma) \\ &= \delta(Q_0 - \{q_i, q_j\}, \sigma) \oplus \delta(q_i, \sigma) \oplus \delta(q_j, \sigma) \\ &= \delta(Q_0, \sigma). \end{aligned}$$

For every word $x \in \Sigma^*$ with $x = x_0 x_1 \dots x_{k+1}$ where $x_k \in \Sigma$ and $q_i \in \delta(Q_0, x_0 x_1 \dots x_k)$ we have that $q_j \in \delta(Q_0, x_0 x_1 \dots x_k)$. So that,

$$\begin{aligned} \delta'(Q'_0, x) &= \delta'(\delta'(Q_0, x_1 x_1 \dots x_k), x_{k+1}) \\ &= \delta'(\delta'(Q_0, x_1 x_1 \dots x_k) - \{q_j\}, x_{k+1}) \oplus \delta'(q_j, x_{k+1}) \\ &= \delta'(\delta'(Q_0, x_1 x_1 \dots x_k) - \{q_j\}, x_{k+1}) \oplus \delta(\{q_i, q_j\}, x_{k+1}) \\ &= \delta'(\delta(Q_0, x_1 x_1 \dots x_k) - \{q_i, q_j\}, x_{k+1}) \oplus \delta(\{q_i, q_j\}, x_{k+1}) \\ &= \delta(\delta(Q_0, x_1 x_1 \dots x_k), x_{k+1}). \end{aligned}$$

Thus $\delta(Q_0, x) = \delta'(Q_0, x)$ so that $\delta(Q_0, x) \cap F' = \delta'(Q_0, x) \cap F'$. As we assumed that, without loss of generality, that if $q_i \in F$ then $q_j \in F$ then with $F' = F - \{q_i\}$ we have that $|\delta(Q_0, x) \cap F| > 0 \Leftrightarrow |\delta(Q_0, x) \cap F'| > 0$.

This means that $\mathcal{L}(M) = \mathcal{L}(M')$. □

As with the initial-useless and final-useless states, redundant states can be detected through the use of the subset construction. The algorithm we present here takes as input a state q and returns the set of states which are redundant with respect to the given state. This is done by processing each individual state in the DFA equivalent to the $\oplus$-FA. If the state contains q then we take the intersection between the state and a set representing the states which are redundant with respect to q. After this initial step we process each state in the DFA equivalent to the $\oplus$-FA again. If the state being processed contains a state in the set of possibly redundant states but does not contain q then it is removed.

Algorithm 5.26 (Redundant States)

Let M be a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ and state $q_i \in Q$. The algorithm outputs the set I of all states redundant with respect to q_i.

1. *Construct $M' \leftarrow \text{DFA}(M) = (P, \Sigma, \delta', p_0, \oplus)$ using on the fly subset construction.*

2. *Construct $I \leftarrow Q - \{q_i\}$.*

3. For every $p \in P$ with $q_i \in p$ do

$$I \leftarrow I \cap p.$$

4. For every $q \in I$ do

If $\exists\ p \in P$ so that $q \in p$ and $q_i \notin p$ then

$$I \leftarrow I - \{q\}.$$

5. Return I.

Algorithm 5.26 finds all the states redundant with respect to a given state by first finding every state which is set active when the given state is active. Then these states are checked to see that whenever they are set active the given state is set active. If this is true these states are redundant with respect to each other.

Equivalent States

Equivalent states are states which have identical successors. That is, in a $*$-FA $M = (Q, \Sigma, \delta, Q_0, F, *)$ a state $q_i \in Q$ is equivalent to a state $q_j \in Q$ if for every word $x \in \Sigma^*$

$$\delta(q_i, x) \cap F \neq \emptyset \ \Leftrightarrow\ \delta(q_j, x) \cap F \neq \emptyset.$$

Note 10 *It is important to note in the definition of equivalent states that the predecessors of the two states do not have to be accepted by the same final states. That is, with $x \in \Sigma^*$ if $\delta(q_i, x) \cap F = \{q\}$ then $\delta(q_j, x) \cap F$ does not have to be equal to $\{q\}$. As long as $\delta(q_j, x) \cap F \neq \emptyset$ it satisfies the definition of equivalence.*

We have already shown how initial-useless, final-useless and redundant states can be reduced in a $\oplus$-FA. However, not all equivalent states in a $\oplus$-FA are reducible. We first show an example of equivalent states which are reducible, and then an example of equivalent states which are irreducible. We then describe a subset of equivalent states which are reducible in a $\oplus$-FA.

Example 5.27 (Reducible Equivalence)
Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, \{q_0\}, \{q_4\}, \oplus)$ with δ given by (see also figure 5.10):

δ	0	1
q_0	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_3, q_4\}$	$\{q_0, q_1\}$
q_2	$\{q_3, q_4\}$	$\{q_0, q_1\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\emptyset$	$\emptyset$

.

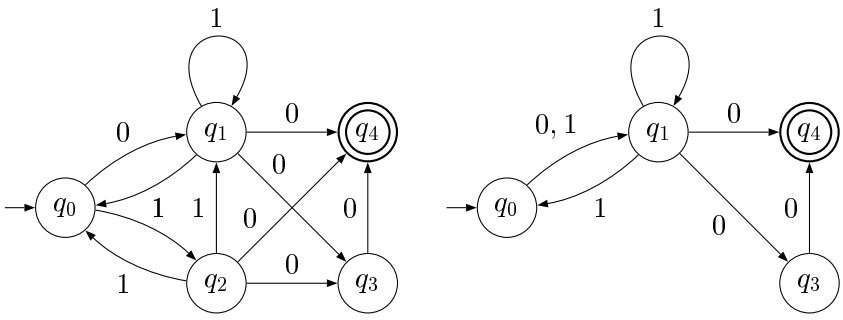

Figure 5.10: Example 5.27: Left M; Right M'

Examining M, we can see that the states q_1 and q_2 are equivalent, as they have identical transitions and so have identical successors. We attempt to reduce the $\oplus$-FA by combining these two states so that we have only one state with the shared predecessor.

This is done by removing one of the states, in this case q_2, and combining the incoming transitions of the equivalent states using the $\oplus$ operator. We remove the outgoing transitions of q_2 and leave the outgoing transitions of q_1 intact. This is done as follows:

The $\oplus$-FA $M' = (\{q_0, q_1, q_3, q_4\}, \{0, 1\}, \delta', \{q_0\}, \{q_4\}, \oplus)$ is constructed from M where δ' is given by (see also figure 5.10):

δ'	0	1
q_0	$\{q_1\}$	$\{q_1\}$
q_1	$\{q_3, q_4\}$	$\{q_0, q_1\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\emptyset$	$\emptyset$

.

It is easy to verify that $\mathcal{L}(M') = \mathcal{L}(M)$ by comparing their equivalent minimal DFAs.

However, as shown in the following example, not all equivalent states are reducible in a $\oplus$-FA.

Example 5.28 (Irreducible Equivalence)

Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_0\}, \{q_2, q_3\}, \oplus)$ with δ given by (see also figure 5.11):

δ	0	1
q_0	$\{q_0, q_1\}$	$\{q_2\}$
q_1	$\{q_0\}$	$\{q_3\}$
q_2	$\emptyset$	$\emptyset$
q_3	$\emptyset$	$\emptyset$

.

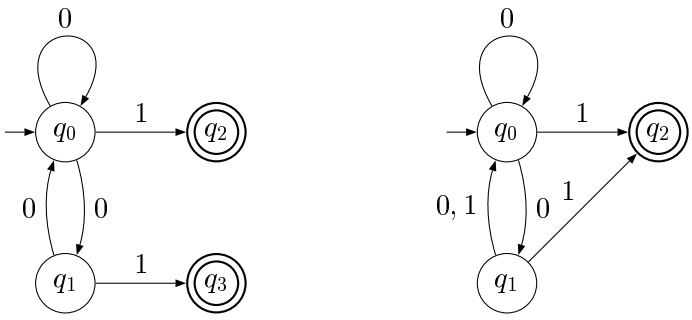

Figure 5.11: Example 5.28: Left M; Right M'

Examining M we note that the states q_2 and q_3 have no outgoing transitions. Also as these states are both final they accept the empty input ϵ. Thus they have identical successors, namely $\{\epsilon\}$. We can attempt to reduce M by combining the equivalent states q_2 and q_3 as in example 5.27. This results in the $\oplus$-FA M' below.

The $\oplus$-FA $M' = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta', \{q_0\}, \{q_2\}, \oplus)$ is constructed from M where δ' is given by (see also figure 5.11):

δ'	0	1
q_0	$\{q_0, q_1\}$	$\{q_2\}$
q_1	$\{q_0\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

.

It is easy to verify that in M the word 01 is accepted. However, it is not accepted in M'. This implies that $\mathcal{L}(M) \neq \mathcal{L}(M')$.

In example 5.28 above the states q_0 and q_1 are also equivalent and can be reduced. The example shows us an interesting property that final states may be equivalent but irreducible. Similarly, other states may be equivalent but irreducible depending on the interactions of the transitions to and from the equivalent states. In the example the equivalent states q_2 and q_3 were irreducible due to the interaction of the parents q_0 and q_1. Combining the states q_2 and q_3 caused the final state to become inactive from the active state set $\{q_0, q_1\}$ where at least one final state should have been active. In fact no modification of the transitions on alphabet symbol 1 from the states q_0 and q_1 will be able to allow for the reduction of the equivalent states.

By experimentation we have identified three conditions of interactions of transitions to and from equivalent states which cause the equivalent states to be irreducible. In these conditions the equivalent states are irreducible barring other influences such as if the states are redundant, and so on.

In a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with equivalent states $q_i, q_j \in Q$, the states can become irreducible under any one of the following conditions:

- The first case is a generalization of example 5.28. Given two equivalent states, q_i and q_j, where $q_i, q_j \in F$. If there exists a word $y \in \Sigma^*$, so that $\{q_i, q_j\} \subseteq \delta(Q_0, y)$ then q_i and q_j are irreducible. The reason for this is that combining the incoming transitions using the $\oplus$ operator results in the remaining state not being set active on the word y.

- Similarly, for every $x \in \Sigma^*$, $\delta(q_i, x) \cap \delta(q_j, x) \neq \emptyset$ and a word $y \in \Sigma^*$ exists so that $\{q_i, q_j\} \subseteq \delta(Q_0, y)$. These states are irreducible as by removing q_i we have that $q_j \notin \delta'(Q_0, y)$, possibly removing some words from the accepted language.

- If the words $x_0, x_1 \in \Sigma^*$ exist so that $x_0 = y_0 z_0$ and $x_1 = y_1 z_1$ with $y_0 \neq y_1$ so that $q_i \in \delta(Q_0, y_0)$ and $q_j \in \delta(Q_0, y_1)$ and $\delta(q_i, z_0) \cap \delta(q_j, z_1) \neq \emptyset$. These states are irreducible as by removing q_i we have that $\delta(q_i, z_0) \cap \delta(q_j, z_1) \subseteq \delta'(Q_0, x)$ where $\delta(q_i, z_0) \cap \delta(q_j, z_1) \not\subseteq \delta(Q_0, x)$.

Note 11 *We repeat here that, as briefly mentioned above, the equivalent states mentioned above may be reducible due to external factors. For example the states could be reducible due to conditions unrelated to equivalence or due to the interaction of reachable active state sets which are difficult to track.*

It is for these reasons that we have been unable to formally define or detect all reducible equivalent states in a $\oplus$-FA. However, we will discuss forms of equivalent states which are reducible during the algorithms we devise later for the reduction and minimization of $\oplus$-FAs.

Contained

Contained states are states whose predecessor and successor are contained within another state's predecessor and successor respectively, that is in a $*$-FA $M = (Q, \Sigma, \delta, Q_0, F, *)$ a state $q_i \in Q$ is contained in a state $q_j \in Q$ if for every word $x \in \Sigma^*$,

$$\begin{aligned} x \in pr(q_i) &\Leftrightarrow x \in pr(q_j), \text{ and} \\ x \in sc(q_i) &\Leftrightarrow x \in sc(q_j). \end{aligned}$$

Contained states can be seen as states which are both redundant and equivalent with respect to another state. However, this relationship does not need to be true both ways. As contained states share properties with equivalent states, they also share the problems inherent with the equivalent states in a $\oplus$-FA as discussed in the previous section. For this reason we do not discuss the reduction of contained states, but instead treat it as a special case of equivalence where a state q_i's successor forms a subset of another state q_j's successor, that is $sc(q_i) \subseteq sc(q_j)$.

5.2.2 Language Construction

Language construction algorithms deal with the construction of a FA from the language it accepts. The algorithm of dictionary minimization by Daciuk, Watson and Watson [4] is a specialized form of language reconstruction, which involves inserting every possible word accepted by the dictionary into the FA one word at a time. However, as most FAs have an infinite number of words due to cyclic transitions, this algorithm becomes unwieldy for general FAs.

Most language reconstruction algorithms deal instead with the regular expression[15] of the language. Generally, using rules governing regular expressions, the original regular expression is first split into smaller and simpler regular expressions. Individual FAs are then constructed for each of these simpler regular expressions. These FAs are then combined, in accordance to the method in which the regular expression was split, to construct a FA which accepts the language represented by the original regular expression.

In this section we begin by describing the factors involved in constructing the individual $\oplus$-FAs representing the various simple forms of regular expressions. We use these examples to illustrate the fact that the language reconstruction methods used for the conventional NFAs may not be applicable to $\oplus$-FAs.

Regular Expressions

A typical language reconstruction algorithm is presented in [15], where a FA $M = (Q, \Sigma, \delta, Q_0, F)$ can be constructed from a regular expression x with alphabet Σ as follows:

- Where $x = \sigma$ with $\sigma \in \Sigma$, we construct the FA M which accepts the regular expression x as $M = (\{q_0, q_1\}, \Sigma, \delta, \{q_0\}, \{q_1\})$ with $\delta(q_0, \sigma) = \{q_1\}$.

- For $x = y|z$ with y and z regular expressions accepted by the FA $M_y = (Q_y, \Sigma, \delta_y, Q_{y0}, F_y)$ and $M_z = (Q_z, \Sigma, \delta_z, Q_{z0}, F_z)$ respectively, we construct $M = (\{q_0\} \cup Q_y \cup Q_z, \Sigma, \delta, \{q_0\}, F_y \cup F_z)$ with δ given by

$$\begin{array}{lcll} \delta(q_0, \epsilon) & = & Q_{y0} \cup Q_{z0}, & \\ \delta(q, \sigma) & = & \delta_y(q, \sigma), & \text{for } q \in Q_y \text{ and every } \sigma \in \Sigma, \text{ and} \\ \delta(q, \sigma) & = & \delta_z(q, \sigma), & \text{for } q \in Q_z \text{ and every } \sigma \in \Sigma. \end{array}$$

- For $x = yz$ with y and z regular expressions accepted by the FA $M_y = (Q_y, \Sigma, \delta_y, Q_{y0}, F_y)$ and $M_z = (Q_z, \Sigma, \delta_z, Q_{z0}, F_z)$ respectively,

[15] See convention A.8, page 170.

we construct $M = (Q_y \cup Q_z, \Sigma, \delta, Q_{y0}, F_z)$ with δ given by

$$\begin{array}{lcll} \delta(q,\sigma) & = & \delta_y(q,\sigma), & \text{for } q \in Q_y, \text{ and every } \sigma \in \Sigma, \\ \delta(q,\epsilon) & = & Q_{z0}, & \text{for } q \in F_y, \text{ and} \\ \delta(q,\sigma) & = & \delta_z(q,\sigma), & \text{for } q \in Q_z \text{ and every } \sigma \in \Sigma. \end{array}$$

- For $x = y^*$ with y a regular expression accepted by the FA $M_y = (Q_y, \Sigma, \delta_y, Q_{y0}, F_y)$, we construct $M = (Q_y, \Sigma, \delta, Q_{y0}, F_y)$ with δ given by

$$\begin{array}{lcll} \delta(q,\sigma) & = & \delta_y(q,\sigma), & \text{for } q \in Q_y, \text{ and every } \sigma \in \Sigma, \text{ and} \\ \delta(q,\epsilon) & = & Q_{y0}, & \text{for } q \in Q_{y0}. \end{array}$$

Note 12 *The construction techniques above based on regular expressions make use of transitions on the ϵ symbol, as in the case of $x = y|z$. These transitions may in fact be removed from a FA and replaced with all the transitions corresponding to the transitions of the active state set of the ϵ transition. That is in a FA M with state set Q where $q \in Q$, $Q' \subseteq Q$, $\delta(q,\epsilon) = Q'$, and for $\sigma \in \Sigma$, we have that*

$$\delta(q, \epsilon\sigma) \;=\; \delta(Q', \sigma).$$

In this article we prefer to use FA without ϵ transitions and thus will remove any instances of ϵ transitions as described in this note.

Note 13 *It is important to note that this method of ϵ-removal does not work for the case of $\oplus$-FAs. The reason for this is the interaction of states in the $\oplus$-FAs.*

Our goal is to be able to construct a minimal $\oplus$-FA from the given regular expression of the language. Before we can do this we need to be able to construct a $\oplus$-FA from a regular expression.

The construction techniques described above generate NFAs which accept the language represented by a given regular expression, but the NFAs are not necessarily minimal. We use these techniques rather than minimal construction techniques as they are simpler to follow, implement and adapt for our uses in $\oplus$-FAs.

We begin by adapting the language construction techniques for NFAs to $\oplus$-FA by combining $\oplus$-FAs which accept languages represented by given regular expressions. As in the case of conventional NFAs, we need techniques to construct and combine $\oplus$-FAs with languages represented by the regular expressions σ, $y|z$, yz and y^* where y and z are regular expressions and σ is an alphabet symbol.

It is easy to see that the $\oplus$-FA which accepts the language represented by the regular expression σ is simply $M = (\{q_0, q_1\}, \{\sigma\}, \delta, \{q_0\}, \{q_1\}, \oplus)$.

Furthermore, with $\delta(q_0, \sigma) = \{q_1\}$ and $\delta(q_1, \sigma) = \emptyset$ we have that $\mathcal{L}(M) = \{\sigma\}$.

The first regular expression we take a serious look at is $y|z$. The construction technique from [15] for NFAs for this case keeps the two NFAs separate from each other and provides an initial state with an ϵ transition to the original initial states of the original NFAs. By adjusting this technique for $\oplus$-FAs we can produce the desired result. However, as we will show later, the other language construction techniques do not provide simple conversion solutions.

Example 5.29 (Language Reconstruction: $y|z$)

Let $y = (00)^*$ and $z = (000)^*$ be the regular expressions representing the languages accepted by the $\oplus$-FAs $M_y = (\{y_0, y_1\}, \Sigma, \delta_y, \{y_0\}, \{y_1\}, \oplus)$ and $M_z = (\{z_0, z_1, z_2\}, \Sigma, \delta_z, \{z_0\}, \{z_2\}, \oplus)$ respectively, where δ_y and δ_z are given by (see also figure 5.12)

δ_y	0
y_0	y_1
y_1	y_0

and

δ_z	0
z_0	z_1
z_1	z_2
z_2	z_0

.

Using the language construction technique from [15] with reference to Note 12, a $\oplus$-FA M can be constructed which accepts the language $y|z$ with $M = (\{q_0, y_0, y_1, z_0, z_1, z_2\}, \Sigma, \delta, \{q_0\}, \{y_1, z_2\}, \oplus)$ with δ given by (see also figure 5.12):

δ	0
q_0	$\{y_1, z_1\}$
y_0	$\{y_1\}$
y_1	$\{y_0\}$
z_0	$\{z_1\}$
z_1	$\{z_2\}$
z_2	$\{z_0\}$

.

As the states of the $\oplus$-FA which accept the sequences y and z are kept disjoint[16] *in the $\oplus$-FA M their transitions cannot interact. Instead the two $\oplus$-FAs run simultaneously in M and so the language $y|z$ is accepted.*

We have examined the case of combining two $\oplus$-FAs whose combined language which corresponds to the regular expression $y|z$. In this case we simply followed the technique used to combine conventional NFAs as given in [15]. Next we examine the case yz. However, as we will show in the following example, this situation is not solved in as simple a fashion as $y|z$.

[16] No transition exists from the one set of states to the other set of states.

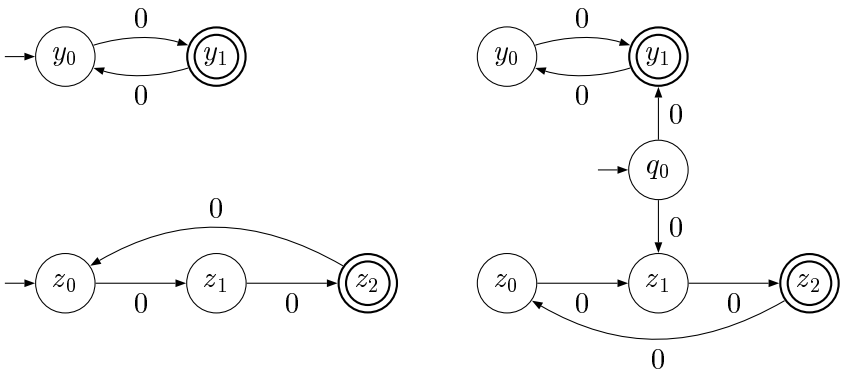

Figure 5.12: Example 5.29: Left M_y and M_z; Right M

Example 5.30 (Language Reconstruction: yz)

Let $y = 01^*$ and $z = 1^*0$ be the regular expressions representing the languages accepted by the $\oplus$-FA $M_y = (\{y_0, y_1\}, \{0,1\}, \delta_y, \{y_0\}, \{y_1\}, \oplus)$ and $M_z = (\{z_0, z_1\}, \{0,1\}, \delta_z, \{z_0\}, \{z_1\}, \oplus)$ where δ_y and δ_z are given by (see also figure 5.13)

δ_y	0	1
y_0	y_1	$\emptyset$
y_1	$\emptyset$	y_1

and

δ_z	0	1
z_0	z_1	z_0
z_1	$\emptyset$	$\emptyset$

.

It is important to note that we construct the transition function, δ, using the following rule, where $y_i \in Q_y$ and $\sigma \in \Sigma$.

If $\delta_y(y_i, \sigma) \cap F \neq \emptyset$ then $\delta(y_i, \sigma) = \delta_y(y_i, \sigma) \oplus Q_{z0}$.

This means that in this example $\delta(y_0, 0) = \{y_1\} \oplus \{z_0\}$. Similarly, $\delta(y_1, 1) = \{y_1\} \oplus \{z_0\}$.

Constructing the $\oplus$-FA M for the language $yz = 01^*1^*0$ as for NFAs we have that $M = (\{y_0, y_1, z_0\}, \{0,1\}, \delta, \{y_0\}, \{z_0\}, \oplus)$ with δ given by (see also figure 5.13)

δ	0	1
y_0	$\{y_1, z_0\}$	$\emptyset$
y_1	$\emptyset$	$\{y_1, z_0\}$
z_0	$\{z_1\}$	$\{z_0\}$
z_1	$\emptyset$	$\emptyset$

Examining M we have that

$$\begin{aligned}
\delta(y_0, 0) &= \{y_1, z_0\}, \\
\delta(y_0, 01) &= \delta(y_0, 0(11)^*1) = \{y_1\}, \\
\delta(y_0, 011) &= \delta(y_0, 0(11)^*) = \{y_1, z_0\}, \\
\delta(y_0, 010) &= \delta(y_0, 0(11)^*10) = \emptyset, \text{ and} \\
\delta(y_0, 0110) &= \delta(y_0, 0(11)^*0) = \{z_1\}.
\end{aligned}$$

This results in $\mathcal{L}(M) = 0(11)^*0 \neq 01^*(1^*0) = yz$.

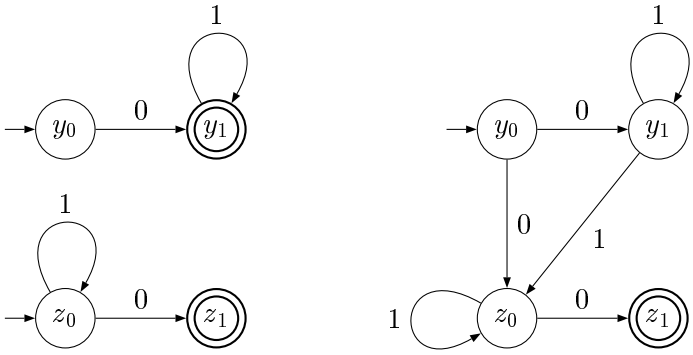

Figure 5.13: Example 5.30: Left M_y and M_z; Right M

As can be seen in example 5.30, the language accepted by M is not the intended language. The reason for this is that all final states in M_y have transitions to the states with initial states as parents in M_z. Thus whenever a final state in M_y is active, it will affect the active states in M_z. Due to the nature of the $\oplus$-operator, these transitions from the final state of M_y will affect the active state sets in M_z. In the case of NFAs, this is not an issue as the transitions cannot set states inactive.

This situation may be rectified by generating a new set of states representing M_z for each instance of a final state in M_y being set active. This situation may be improved by keeping track of the active state sets in M_z and keeping these disjoint from each other.

However, as there are 2^n possible active state sets in M_z where $n = |Q_z|$ this is not a practical solution.

It is important to note that although solutions may be found to rectify the error in single cases such as example 5.30, we are looking for a general case solution.

The next regular expression we examine is that of y^*.

Example 5.31 (Language Reconstruction: y^*)

Let $y = 11^*$ be the regular expression representing the language accepted by the $\oplus$-FA $M_y = (\{y_0, y_1\}, \{1\}, \delta_y, \{y_0\}, \{y_1\}, \oplus)$ with δ_y given by (see also figure 5.14)

δ_y	1
y_0	$\{y_1\}$
y_1	$\{y_1\}$

.

A $\oplus$-FA M can be constructed for the language $y^* = (11^*)^*$ as for NFAs in [15]. We have that $M = (\{y_0, y_1\}, \{1\}, \delta, \{y_0\}, \{y_1\}, \oplus)$ with δ given by (see also figure 5.31)

δ	1
y_0	$\{y_1\}$
y_1	$\{y_0, y_1\}$

.

Figure 5.14: Example 5.31: Left M_y; Right M

Examining M we have that

$$\begin{aligned} \delta(y_0, 1) &= \{y_0, y_1\} \\ \delta(y_0, 11) &= \emptyset \end{aligned}$$

Therefore, $\mathcal{L}(M) = 1 \neq (11^*)^* = y^*$.

As with example 5.30 the constructed $\oplus$-FA M in example 5.31 does not accept the intended language. As with example 5.30 the reason for this is the interaction between the final and initial states and the $\oplus$ operator.

The solution to this problem would be to create an instance of M_y for every possible repetition discarding redundancies, but as discussed for example 5.30 the number of resultant states is impractical.

The complexity of transforming the yz and y^* regular expressions to $\oplus$-FAs as discussed in examples 5.30 and 5.31 does not bode well for the use of language reconstruction for $\oplus$-FAs. However, in the cases above we were attempting to construct the $\oplus$-FA in conditions best suited for NFAs.

As the languages of FAs are firmly based upon regular expressions, which do not translate easily to the format of $\oplus$-FAs, language reconstruction does not present a viable option for minimization with the current knowledge of $\oplus$-FAs.

5.2.3 Transformation Construction

Transformation construction algorithms use various transformations, such as Kameda and Weiner's NFA minimization algorithm through the intersection rule [11], and Brzozowski's DFA minimization algorithm [2]. As in Brzozowski's method which uses only the subset and dual constructions, the transformations applied for minimization purposes do not always need to be able to detect the removable states. However, the detection of removable states often leads to an increase in computational complexity as with Kameda and Weiner and the states maps[17].

[17] See section 4.3.1, page 61.

Kameda and Weiner's intersection rule allows for the construction of an NFA, given a DFA and a subset assignment. However, the resultant NFA is not always equivalent to the DFA unless a valid subset assignment is used. Fortunately, as shown by Kameda and Weiner, the subset assignment constructed using a cover map is always valid. Kameda and Weiner's cover maps were constructed by making use of the equivalence DFAs, constructed through subset construction, of both the original NFA and its dual.

We begin by discussing how Kameda and Weiner's algorithm of NFA reconstruction can be modified for use on $\oplus$-FAs.[18] After this we look at the possibilities of constructing a cover map to guarantee the construction of a minimal $\oplus$-FA. To apply this approach, we need the concept of the dual of a $\oplus$-FA, which we shall discuss in detail.

Subset Reconstruction

We begin by generalizing Kameda and Weiner's intersection rule which was constructed for NFAs ($\cup$-FAs) to the general operator, that is $*$-FAs. Using this generalization we discuss how the constructed algorithm would function for $\oplus$-FAs.

To briefly recap, the intersection rule can be seen as an inverse to the subset construction operation[19] which constructs an equivalent DFA from an NFA. Given a DFA the intersection rule constructs an NFA using a subset assignment which maps states in the DFA to states in the NFA. The transitions in the NFA are constructed using the transitions in the DFA and the subset assignment. Thus the constructed NFA is dependant entirely upon the transitions of the DFA and the subset assignment. However, where the subset construction method guarantees an equivalent DFA, the intersection rule does not guarantee an equivalent NFA. A subset assignment which produces an equivalent NFA is defined as valid.

Formally, the Intersection Rule, I, is defined on a DFA M' and the subset assignment $< Q, f >$ where $M' = (P, \Sigma, \delta', p_0, F')$ and Q is the set of NFA states with

$$f : P \to 2^Q.$$

Using this notation we define the intersection rule generalization, which we call *Subset Reconstruction*, as follows:

Definition 5.32 (Subset Reconstruction)
Given a DFA $M' = (P, \Sigma, \delta', p_0, F')$ and the subset assignment $< Q, f >$ we can construct a $*$-FA $M = (Q, \Sigma, \delta, Q_0, F, *)$ as follows:

[18] At the end of this chapter we discuss how this and other methods can be modified for use on other $*$-FAs such as $\cap-$FAs. See section 5.3, page 137.

[19] See section 1.3, page 13.

- *The initial state set of M is constructed from the initial state of M' as*

$$Q_0 \;=\; f(p_0).$$

- *The final state set F of M is constructed from the final states of M' as follows:*

$$q \in F \text{ if } q \in Q \text{ and for all } p \in P \text{ with } q \in f(p),\ p \in F',$$

  and for all $p \in F'$ at least one state $q \in f(p)$ must satisfy $q \in F$.

- *The transition function is constructed on a state by state basis. A state q contains a transition to another state in the $*$-FA if the transitions of each state in the DFA which contains q is satisfied by the transition. Thus δ is constructed from δ' with $q, r \in Q$ and $\sigma \in \Sigma$ as follows:*

  $r \in \delta(q, \sigma)$ if for all $p \in P$ where $q \in f(p)$ and $p' = \delta'(p, \sigma)$ we have that $r \in \delta(f(p'), \sigma)$.

Simply put, the transition function in M must ensure that its transitions satisfy the transitions of M'. This is done by comparing the transitions in M' to the mappings of the transitions in M and the operator that M uses. As the definition deals with the general case, we cannot simplify the above requirement. However, it can be simplified during the construction of a specified operator as in the case of Kameda and Weiner for NFAs using the intersection rule.

It is also important to note that the subset reconstruction algorithm will not produce an equivalent $*$-FA if any of the three construction rules in definition 5.32 are not followed. The initial state set is easily verified. In the case of the final state set, every final state in M' must have a final state of M in its mapping and every state in M' which has a final state of M in its mapping must be final.

With the subset reconstruction algorithm of definition 5.32 in hand we can construct the subset reconstruction algorithm in terms of $\oplus$-FAs.

We can construct a $\oplus$-FA from a given DFA as follows:

Definition 5.33 (Subset Reconstruction: $\oplus$-FA)
Let M' be a DFA $M' = (P, \Sigma, \delta', p_0, F')$ and assume a subset assignment $< Q, f >$. We can construct a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ as follows:

- $Q_0 = f(p_0)$.
- $q \in F$ if $q \in Q$ and for all $p \in P$ with $q \in f(p),\ p \in F'$.
- *For $q \in Q$ and $\sigma \in \Sigma$*

– *if* $\exists p \in P$ *so that* $f(p) = \{q\}$ *then*

$$\delta(q, \sigma) \;=\; f(\delta'(p, \sigma)),$$

– *else if* $\exists P_i \subseteq P$ *so that* $\oplus_{p \in P_i} f(p) = \{q\}$ *then*

$$\delta(q, \sigma) \;=\; \oplus_{p \in P_i} f(\delta'(p, \sigma)),$$

– *else if* $\exists p \in P$ *and* $q_i \in Q$ *so that* $\forall q_i \in Q_i$, $\delta(q_i, \sigma)$ *is known and* $f(p) \oplus Q_i = \{q\}$ *then*

$$\delta(q, \sigma) \;=\; f(\delta'(p, \sigma)) \oplus Q_i.$$

Note 14 *The first two points of definition 5.33 construct the initial state set and the final state set of the* $\oplus$*-FA. The third point describes the construction of the transition function. This is done for each state* q *in the* $\oplus$*-FA individually in three possible methods.*

The first method occurs when a state in the DFA p *maps directly into a state in the* $\oplus$*-FA. That is,*

$$f(p) = \{q\}.$$

In this case the transition function of q *in the* $\oplus$*-FA is simply the mapping of the transition function of* p *in the DFA. Thus*

$$\delta(q, \sigma) \;=\; f(\delta'(p, \sigma)) \oplus Q_i.$$

It is important to note that in the second case we refer to cases of two states in the DFA. However, in the definition we refer to sets of states. We refer in this note to the case of two states at it is easier to follow.

The second method occurs when two states, p_i, p_j*, exist in the DFA of which the symmetric difference of the mappings is the state* q*. That is,*

$$f(p_i) \oplus f(p_j) \;=\; \{q\}.$$

From here we construct the transition function of q *in a similar way to the first method. The only difference is we that we apply the symmetric difference operator to the mappings of the transitions of the two states involved. That is,*

$$\delta(q, \sigma) \;=\; f(\delta'(p_i, \sigma)) \oplus f(\delta'(p_j, \sigma)).$$

The third method occurs when a state p *in the DFA maps to a set of states* $f(p) = Q'$ *of the* $\oplus$*-FA with* $q \in Q'$*. If the transitions of every* $\oplus$*-FA state* $q' \in (Q' - \{q\})$ *is known, then the transitions of the state* q *can be calculated by taking the symmetric difference of the transitions of the set* $(Q' - \{q\})$ *and the mappings of the transitions of the state* p*. That is,*

$$\delta(q, \sigma) \;=\; \delta(Q' - \{q\}, \sigma) \oplus f(\delta'(p, \sigma)).$$

It is also important to note that the various methods can be combined as necessary. For example in the case where a pair of states exist in the DFA of which the symmetric difference of their mappings consist of known states and one unknown state, the transitions of the unknown state can be calculated by combining the second and third methods.

Before presenting an example of definition 5.33 we show that the definition can indeed produce a minimal equivalent $\oplus$-FA from a given $\oplus$-FA.

Theorem 5.34 *A minimal $\oplus$-FA can always be constructed through the use of $\oplus$-FA subset reconstruction from definition 5.33.*

Proof We must show that for any minimal $\oplus$-FA at least one combination of an equivalent DFA and a valid subset assignment exists for which the $\oplus$-FA subset reconstruction will result in the construction of a minimal $\oplus$-FA We show this by reconstructing a minimal $\oplus$-FA from the equivalent DFA constructed through on the fly subset construction from the $\oplus$-FA.

Let $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ be a minimal $\oplus$-FA. We construct $M' = \text{DFA}(M) = (P, \Sigma, \delta', p_0, F')$ through on-the-fly subset construction. Let $< Q, f >$ be the natural subset assignment[20] which maps the states from M' to M. We can construct a $\oplus$-FA $M'' = (Q'', \Sigma, \delta'', Q_0'', F'', \oplus)$ through $\oplus$ subset reconstruction using the natural subset assignment.

From the reconstruction algorithm we have that $Q'' = Q$ and from the natural subset assignment we can determine that $Q_0'' = Q_0$. Furthermore, we can determine F'' through the natural subset assignment to be $F'' \supseteq F$. Thus F'' contains all the states of F and possibly more. In the case were $F \subset F''$ there exists at least one state $q \in F''$ where $q \notin F$. It follows that for every $p \in P$ where $q \in f(p)$ that $p \in F'$. The state q is only ever active when a final state is active. It follows that the language accepted by M will not be effected by adding the state q to F.

This leaves only the transition function of M'' to be determined. We wish to show that for every state $q \in Q$ and for each alphabet symbol $\sigma \in \Sigma$ that $\delta(q, \sigma) = \delta''(q, \sigma)$. This can be determined using brute force. However, we present three methods for generating the $\delta''(q, \sigma)$. These methods do not cover every circumstance and so brute force methods may still be required to generate the transition function of M''.

Based no note 14 the transitions of a state q for an alphabet symbol $\sigma \in \Sigma$ can be calculated in the following ways:

- If there exists a state $p \in P$ where $\{q\} = f(p)$ then we have that

$$\delta''(q, \sigma) \;=\; f(\delta'(p, \sigma)).$$

As $f(\delta(p, \sigma)) = \oplus(\delta(q, \sigma))$ it follows that $\delta''(q, \sigma) = \delta(q, \sigma)$.

[20] See page 57 for the definition of the natural subset.

- If there exists a state $p \in P$ where $q \in f(p)$ and for every state $q' \in Q$ with $q' \in f(p)$ and $q \neq q'$ we have that $\delta(q', \sigma)$ is known, then

$$\delta''(q, \sigma) \;=\; f(\delta'(p, \sigma)) \oplus_{p'} \delta''(q', \sigma).$$

  As all $\delta''(q', \sigma)$ are known, it follows that $\delta''(q', \sigma) = \delta(q', \sigma)$. It can thus be shown that

$$\begin{aligned} \delta''(q, \sigma) &= f(\delta'(p, \sigma)) \oplus_{q' \in f(p), q' \neq q} \delta''(q', \sigma) \\ &= \oplus_{q_i \in f(p)} (\delta(q_i, \sigma)) \oplus_{q' \in f(p), q' \neq q} \delta''(q', \sigma) \\ &= \oplus_{q_i \in f(p)} (\delta(q_i, \sigma)) \oplus_{q' \in f(p), q' \neq q} \delta(q', \sigma) \\ &= \delta(q, \sigma). \end{aligned}$$

- If there exists a set of states $P' \subseteq P$ so that $\oplus_{p \in P'} f(p) = q$ then we have that

$$\delta''(q, \sigma) \;=\; f(\oplus_{p \in P'} \delta'(p, \sigma)).$$

  Let $p' = \{\oplus_{p \in P'} f(p)\}$ then we have that $p' = \{q\}$ so that $f(p') = q$. From above and the $\oplus$ operator we have that

$$\delta''(q, \sigma) \;=\; \delta(q, \sigma).$$

If the transitions of a state cannot be determined by any of the above methods, a brute force search through all the possible transitions could be generated for the state. Although not efficient, in time the transition set $\delta''(q, \sigma) = \delta(q, \sigma)$ would be generated. Using this transition set and continuing with the construction methods above in time the transition function of M'' would be constructed so that for all $q \in Q$ and $\sigma \in \Sigma$,

$$\delta''(q, \sigma) = \delta(q, \sigma).$$

It follows that $M'' \equiv M$ and $|Q| = |Q''|$ so that M'' is minimal. □

Example 5.35

Take a DFA $M = (\{p_0, p_1, \ldots, p_{11}\}, \{0, 1\}, \delta', p_0, \{p_4, p_5, p_7, p_8, p_{10}, p_{11}\})$ *with* δ' *given by (see also figure 5.15)*

δ'	0	1
p_0	p_1	p_2
p_1	p_3	p_4
p_2	p_5	p_6
p_3	p_7	p_8
p_4	p_4	p_5
p_5	p_2	p_8

δ'	0	1
p_6	p_9	p_5
p_7	p_{10}	p_4
p_8	p_0	p_6
p_9	p_{11}	p_3
p_{10}	p_8	p_2
p_{11}	p_6	p_3

.

To construct a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ from M' we need a subset assignment which we choose as $< Q, f >$ where $Q = \{q_0, q_1, q_2, q_3\}$ and f is described as:

p	$f(p)$	p	$f(p)$
p_0	$\{q_0\}$	p_6	$\{q_2\}$
p_1	$\{q_0, q_2\}$	p_7	$\{q_0, q_1, q_3\}$
p_2	$\{q_1\}$	p_8	$\{q_2, q_3\}$
p_3	$\{q_1, q_2\}$	p_9	$\{q_0, q_1\}$
p_4	$\{q_1, q_3\}$	p_{10}	$\{q_0, q_1, q_2, q_3\}$
p_5	$\{q_3\}$	p_{11}	$\{q_0, q_2, q_3\}$

Clearly, $F = \{q_3\}$, and $Q_0 = \{q_0\}$. Also, δ is constructed as follows (The resultant transitions are shown in figure 5.15):

$$
\begin{aligned}
\delta(q_0, 0) &= f(\delta'(p_0, 0)) = f(p_1) = \{q_0, q_2\} \\
\delta(q_0, 1) &= f(\delta'(p_0, 1)) = f(p_2) = \{q_1\} \\
\delta(q_1, 0) &= f(\delta'(p_3, 0)) \oplus f(\delta'(p_6, 0)) = f(p_7) \oplus f(p_9) \\
&= \{q_0, q_1, q_3\} \oplus \{q_0, q_1\} = \{q_3\} \\
\delta(q_1, 1) &= f(\delta'(p_3, 1)) \oplus f(\delta'(p_6, 1)) = f(p_8) \oplus f(p_5) \\
&= \{q_2, q_3\} \oplus \{q_3\} = \{q_2\} \\
\delta(q_2, 0) &= f(\delta'(p_3, 0)) \oplus \delta(q_1, 0) = f(p_7) \oplus \{q_3\} \\
&= \{q_0, q_1, q_3\} \oplus \{q_3\} = \{q_0, q_1\} \\
\delta(q_2, 1) &= f(\delta'(p_3, 1)) \oplus \delta(q_1, 1) = f(p_8) \oplus \{q_2\} \\
&= \{q_2, q_3\} \oplus \{q_2\} = \{q_3\} \\
\delta(q_3, 0) &= f(\delta'(p_5, 0)) = f(p_2) = \{q_1\} \\
\delta(q_3, 1) &= f(\delta'(p_5, 1)) = f(p_8) = \{q_2, q_3\}
\end{aligned}
$$

In example 5.35 we can verify that $\mathcal{L}(M) = \mathcal{L}(M')$ through the subset construction of M, so that $M \equiv M'$. Furthermore M' is the minimal DFA and as $|Q| = \lceil \log_2(|P|) \rceil$, M' must be a minimal $\oplus$-FA accepting the language $\mathcal{L}(M')$.

When constructing δ we used three different algorithms to calculate the transitions of the state q_0, q_1 and q_2. $\delta(q_0, \sigma)$ was calculated using the mappings of the transitions $\delta(p_0, \sigma)$ in M', $\delta(q_1, \sigma)$ was calculated by taking the symmetric difference of the mappings of the transitions $\delta(p_3, \sigma)$ and $\delta(p_6, \sigma)$. This could be done as $f(p_3) \oplus f(p_6) = \{q_1, q_2\} \oplus \{q_2\} = \{q_1\}$. Lastly $\delta(q_2, \sigma)$ was calculated by taking the symmetric difference of the mapping of the transitions $\delta(p_3, \sigma)$ and the already calculated transitions of q_1, as with q_1 this could be done as $f(p_3) \oplus \{q_1\} = \{q_1, q_2\} \oplus \{q_1\} = \{q_2\}$.

By generating the minimal DFA which accepts a language and using a brute force search through the possible subset assignments an equivalent minimal $\oplus$-FA can be constructed. Thus subset reconstruction guarantees results although the computational complexity of such an algorithm is dependant on the number of possible subset assignments. As there are 2^n

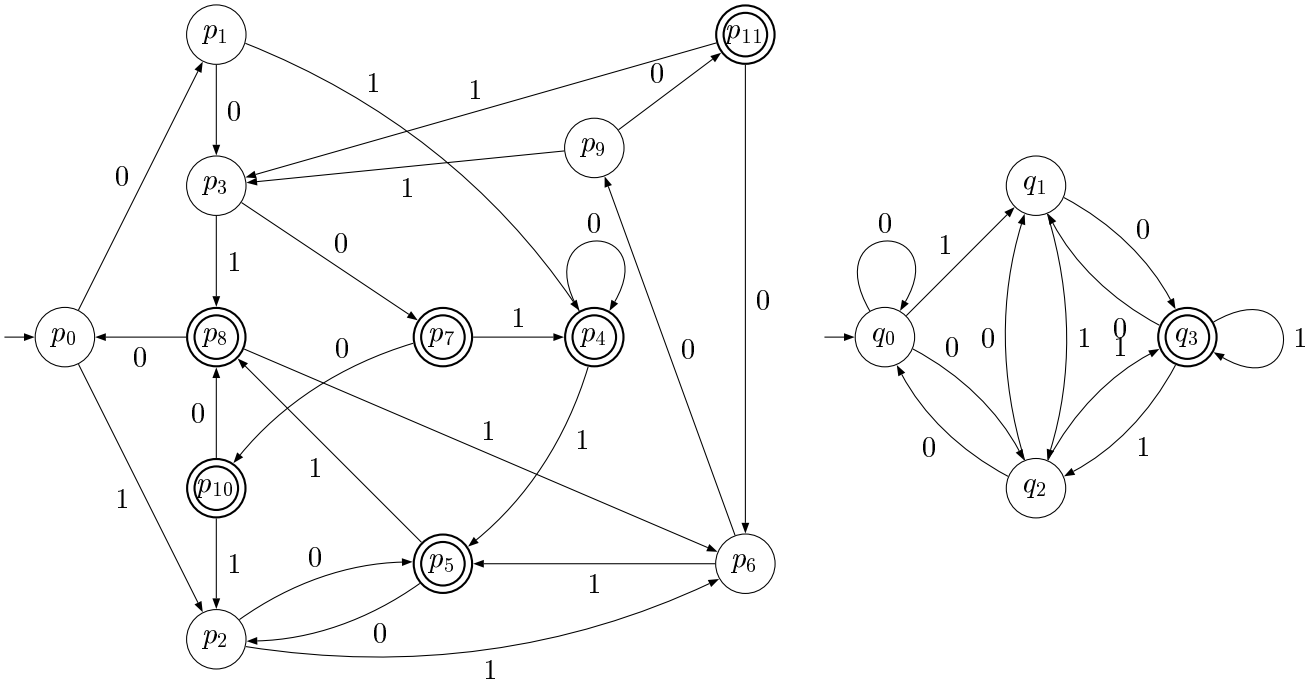

Figure 5.15: Example 5.35: Left M'; Right M

possible subsets of a set of size n, for a minimal DFA with n states, the computational complexity of a brute force search of all the subset assignments for size $\lceil \log_2(n) \rceil$ is $\mathcal{O}(n^n)$ and thus an NP problem.

For the case of NFAs, Kameda and Weiner presented an algorithm to find a subset assignment guaranteed to result in a minimal NFA, this algorithm was based on properties of NFAs as well as their duals. A similar algorithm would be ideal for $\oplus$-FA which we will examine in the next section.

Duals

In chapter 4 we discussed the dual in two separate contexts. The first, in Brzozowski's DFA minimization algorithm, was used to identify equivalent states, as equivalent states in a FA are the redundant states in the dual. The second context in Kameda and Weiner's NFA minimization algorithm was used to construct a cover map which was used to reconstruct an equivalent NFA. The minimal DFAs equivalent to both the NFA and the dual of the DFA are used to construct the cover map. Although subset reconstruction as discussed in the previous section allows us to construct a minimal $\oplus$-FA from a minimal DFA using subset assignments, it does not present an algorithm to generate a valid subset assignment.

In this section we wish to examine if the dual of a $\oplus$-FA can be constructed to be able to generate a valid subset assignment, and also to be able to identify reducible states similarly to NFAs.

From definition 4.17, page 50, we have that the dual $\overleftarrow{M}$ of a FA M

accepts the inverse language of the original FA M, that is

$$\mathcal{L}(\overleftarrow{M}) = \overleftarrow{\mathcal{L}}(M).$$

A first attempt at constructing a dual could be through the use of language construction algorithms. However, the problems inherent with $\oplus$ language reconstruction algorithms as discussed in section 5.2.2 make this solution unrealistic. Instead we examine the construction of duals of NFAs.

For NFAs and DFAs, a dual is constructed by reversing the transition function and swapping the initial and final state sets, that is for $M = (Q, \Sigma, \delta, Q_0, F)$,

$$\overleftarrow{M} = (Q, \Sigma, \overleftarrow{\delta}, F, Q_0), \text{ where } q_i \in \delta(q_j, \sigma) \Leftrightarrow q_j \in \overleftarrow{\delta}(q_i, \sigma),$$

with $q_i, q_j \in Q$ and $\sigma \in \Sigma$.

Due to the nature of the $\oplus$ operator the $\overleftarrow{M}$ as defined above will not generate the inverse language as a state will be set inactive if it is set active twice. For this reason we look not at the actual structure of the dual, but the idea behind it.

The children of a state in the dual are the parents of the state in the original NFA. That is it has a transition to every state that has a transition to it in the original NFA. In turn it has it has an incoming transition from every state it used to have an outgoing transition to.

We attempt to construct a dual for a $\oplus$-FA with that idea in mind by creating transitions from every state to every possible subset of states which can set it as active, where each state in the subset is a parent of the state.

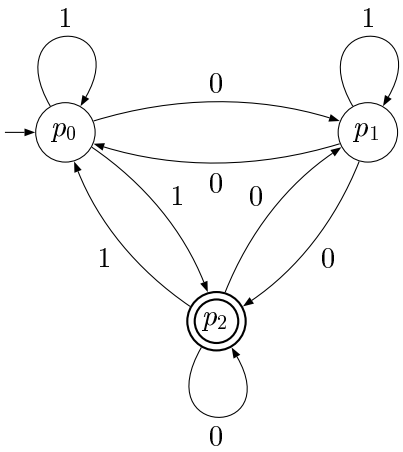

Figure 5.16: Example 5.36: Left M'; Right M

Example 5.36

Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2\}, \Sigma, \delta, \{q_0\}, \{q_1\}, \oplus)$ with δ given by (see also figure 5.16)

δ	0	1
q_0	$\{q_1\}$	$\{q_0, q_2\}$
q_1	$\{q_0\}$	$\{q_1\}$
q_2	$\{q_1\}$	$\{q_0, q_1\}$

.

We wish to construct the dual of M, $\overleftarrow{M} = \{P, \Sigma, \overleftarrow{\delta}, P_0, F_p, \oplus)$. We begin by constructing the state set

5.2.4 State Comparison

We have examined possible $\oplus$-FA minimization algorithms for both language construction and transformation construction, but with limited success. The last algorithm open to us is that of state comparison. However, as described when defining equivalent and contained states the state reduction process is somewhat more complex than that for NFAs. Fortunately the $\oplus$ operator has certain properties which can be used to simplify the detection and removal process of reducible states and transitions in a $\oplus$-FA. We construct a process designed on these properties allowing for the detection and removal of these reducible states without having to use the predecessors or successors of the states in question.

In this section we design and discuss a state transformation process called subset $\oplus$ transformation which we will use to both detect and remove reducible states in a $\oplus$-FA, as well as showing how it can be used to minimize a $\oplus$-FA with NP computational complexity or used to simply reduce a $\oplus$-FA in polynomial time.

Subset $\oplus$ Transformation

When we constructed the transition function in the subset reconstruction algorithm for $\oplus$-FA in section 5.2.3 we made use of the following property of states in a $\oplus$-FA:

In a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ where $Q_i \subseteq Q$ with $q_i \in Q_i$ and $\sigma \in \Sigma$ we have that

$$\delta(q_i, \sigma) \;=\; \delta(Q_i, \sigma) \oplus \delta(Q_i - \{q_i\}, \sigma).$$

In subset reconstruction we used this property to calculate the transitions of a state in the constructed $\oplus$-FA. In this section we use this property in a different way.

The basic idea behind the subset $\oplus$ transformation is to create an equivalent $\oplus$-FA by replacing a state with a state representing a subset of states of which the original state belonged.

This is done as follows:

Definition 5.37 (Subset $\oplus$ Transformation)
Let M be a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ and $Q_i \subseteq Q$ with $q_i \in Q_i$ where if $Q_i \cap F \neq \emptyset$, $q_i \in F$, we generate a $\oplus$-FA $M' = (Q', \Sigma, \delta', Q_0', F', \oplus)$ as follows:

- $Q' = (Q \cup \{s\}) - \{q_i\}$ *where the state s replace q_i,*
- *if $q_i \in Q_0$ then $Q_0' = (Q_0 \oplus Q_i) \oplus \{s\}$ else $Q_0' = Q_0$,*
- *if $q_i \in F$ then $F' = (F/\{q_i\}) \cup \{s\}$ else $F' = F$, and*
- *δ' is defined with $\sigma \in \Sigma$ and $q \in Q'$ as follows:*

$$\delta'(q, \sigma) \;=\; \begin{cases} \delta(q, \sigma) & , \; if \; q \neq s \; and \; q_i \notin \delta(q, \sigma) \\ (\delta(q, \sigma) \oplus Q_i) \oplus \{s\} & , \; if \; q \neq s \; and \; q_i \in \delta(q, \sigma) \\ \delta(Q_i, \sigma) & , \; if \; q = s \; and \; q_i \notin \delta(Q_i, \sigma) \\ (\delta(Q_i, \sigma) \oplus Q_i) \oplus \{s\} & , \; if \; q = s \; and \; q_i \in \delta(Q_i, \sigma). \end{cases}$$

As at most all the transitions of the $\oplus$-FA are involved in the construction of the $\oplus$-FA the computational complexity of the algorithm is $\mathcal{O}(m)$ where m is the number of transitions in M.

Before proving that the resultant $\oplus$-FA is in fact equivalent to the original $\oplus$-FA we show the workings of definition 5.37 with the following example:

Example 5.38
Let M be the $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_0\}, \{q_3\}, \oplus)$ from example 5.35with δ given by (see also figure 5.17)

δ	0	1
q_0	$\{q_0, q_2\}$	$\{q_1\}$
q_1	$\{q_3\}$	$\{q_2\}$
q_2	$\{q_0, q_1\}$	$\{q_3\}$
q_3	$\{q_1\}$	$\{q_2, q_3\}$

.

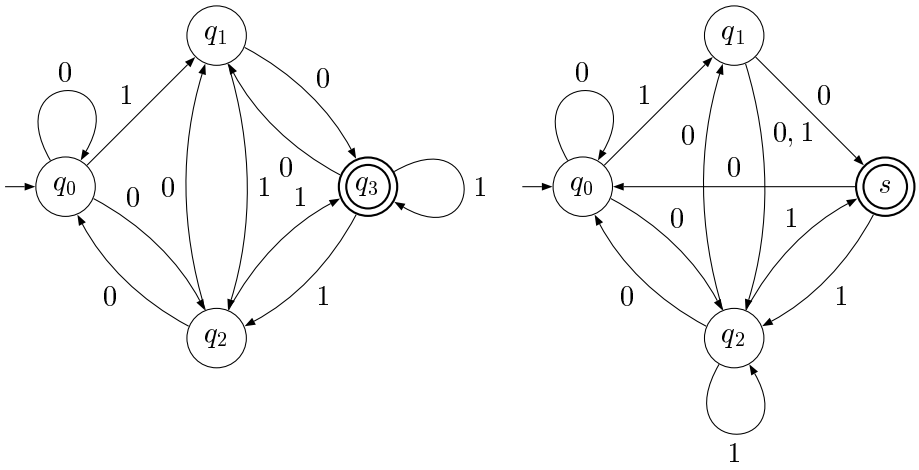

Figure 5.17: Example 5.38: Left M; Right M'

We construct $M' = (\{q_0, q_1, q_2, s\}, \{0,1\}, \delta', \{q_0\}, \{s\}, \oplus)$ *where* $q_i = q_3$ *and* $Q_i = \{q_2, q_3\}$ *with* δ' *given by (see also figure 5.17)*

δ	0	1
q_0	$\{q_0, q_2\}$	$\{q_1\}$
q_1	$\{q_2, s\}$	$\{q_2\}$
q_2	$\{q_0, q_1\}$	$\{q_2, s\}$
s	$\{q_0\}$	$\{q_2\}$

.

Theorem 5.39 *A subset* $\oplus$ *transformed* $\oplus$*-FA* M' *of* M *as defined in definition 5.37 is equivalent to* M*.*

Proof Let M be a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ and $Q_i \subseteq Q$ with $q_i \in Q_i$ where if $Q_i \cap F \neq \emptyset$, $q_i \in F$, and the $\oplus$-FA $M' = (Q', \Sigma, \delta', Q_0', F', \oplus)$ generated as defined in definition 5.37 we need to prove that $M' \equiv M$.

We begin by assigning $\{s\} = Q_i$ and showing that δ', and Q_0' are equal to δ and Q_0, and that F is active if and only if F' is active.

For δ we have for any σ that if

- $q \neq s$ and $q_i \notin \delta(q, \sigma)$ then $\delta'(q, \sigma) = \delta(q, \sigma)$,
- $q \neq s$ and $q_i \in \delta(q, \sigma)$ then

$$\delta'(q, \sigma) \;=\; \delta(q, \sigma) \oplus Q_i \oplus \{s\} \;=\; \delta(q, \sigma) \oplus Q_i \oplus Q_i \;=\; \delta(q, \sigma),$$

- $q = s$ and $q_i \notin \delta(Q_i, \sigma)$ then $\delta'(s, \sigma) = \delta(Q_i, \sigma)$,
- $q = s$ and $q_i \in \delta(Q_i, \sigma)$ then

$$\delta'(s, \sigma) \;=\; \delta(Q_i, \sigma) \oplus Q_i \oplus \{s\} \;=\; \delta(Q_i, \sigma) \oplus Q_i \oplus Q_i \;=\; \delta(Q_i, \sigma),$$

which satisfies the equalities of $\{s\}$ to Q_i and the transition functions.

For the initial state set Q_0', from definition 5.37 if $q_i \notin Q_0$ then $Q_0' = Q_0$ so that Q_0' is equivalent to Q_0, else if $Q_i \in Q_0$ then

$$\delta(Q_0', \sigma) \;=\; \delta(Q_0 \oplus Q_i \oplus \{s\}, \sigma) \;=\; \delta(Q_0 \oplus Q_i \oplus Q_i, \sigma) \;=\; \delta(Q_0, \sigma),$$

so that $\delta'(Q_0', \sigma) = \delta(Q_0, \sigma)$.

For the final state set F, if $q_i \notin F$ then $F' = F$ so that F' is equal to F, else if $q_i \in F$ then $F' = (F \cup \{s\})/\{q_i\}$. We must show that a state in F' is active if and only if a state in F is active. From δ and δ' we have that if

- $q \in F$ is active with $q \notin Q_i$ then $q \in F'$ is active in M',
- $q \in F'$ is active with $q \notin Q_i$ then $q \in F$ is active in M,
- $q \in F$ is active and $q_i \in F$ is not active with $q \in Q_i$ then $q \in F'$ is active in M',
- $q \in F'$ is active and $s \in F'$ is not active with $q \in Q_i$ then $q \in F$ is active in M,
- $q_i \in F$ is active then $s \in F'$ is active in M',
- $s \in F'$ is active then $q_i \in F$ is active in M.

So that a state in F is active only when a state in F' is active.

Thus as M and M' have identical transitions and initial state sets and that F is active if and only if F' is active then $M \equiv M'$. □

The subset $\oplus$ transformation replaces a single state with a state representing a set of states, but this seems to go against the idea of reduction of states. In the following section we will show the uses of this transformation, starting with transformation showing the non-uniqueness of the minimal $\oplus$-FA followed by some construction which simplify the process of minimization of $\oplus$-FAs. After which we will describe how the transformation can be used to reduce a $\oplus$-FA's number of states and transitions before describing how they can be used in various algorithms to reduce or minimize the $\oplus$-FAs.

Structure Transformations

In this section we describe some possible structure transformations for $\oplus$-FA based on subset $\oplus$ transformations. These transformations may not be directly useful in the minimization of $\oplus$-FA, but may instead be useful in the storage and construction of $\oplus$-FA, such as through the use of subset reconstruction.

The first application of the subset $\oplus$-FA is to show that the minimal $\oplus$-FA is not unique. We present only the theorem for this property as

the $\oplus$-FAs M and M', constructed through a subset $\oplus$ transformation, in example 5.38 are minimal[21].

Theorem 5.40 *A minimal $\oplus$-FA is not unique.*

Proof Let M be a minimal $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ then we construct an equivalent $\oplus$-FA $M' = (Q', \Sigma, \delta', Q_0', F', \oplus)$ using the subset $\oplus$ transformation on the subset $Q_i \subseteq Q$ and state $q_i \in Q$ as described in definition 5.37.

We have that $Q' = Q \oplus \{q_i\} \oplus \{s\}$ so that $|Q'| = |Q|$, and furthermore from theorem 5.39 $Q \equiv Q'$. As $|Q'| \leq |Q|$ and M is minimal, the $\oplus$-FA M' must be minimal, thus the minimal $\oplus$-FA is not unique. □

The subset $\oplus$ transformation not only proves that minimal $\oplus$-FA are not unique but also generates equivalent minimal $\oplus$-FAs.

For the next application of the subset $\oplus$ transformation we define the *single initial state transformation* as the subset $\oplus$ transformation on a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ where the subset chosen is the initial state set, that is $Q_i = Q_0$, and the state replaced is an initial state of M. We first describe such an instance in an example before discussing it further.

Example 5.41 (Initial State Set)

Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_0, q_1\}, \{q_3\}, \oplus)$ with δ given by (see also figure 5.18)

δ	0	1
q_0	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_0, q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\{q_1, q_2\}$
q_3	$\{q_0\}$	$\{q_3, q_2\}$

.

We can construct an equivalent $\oplus$-FA M' with exactly one initial state set using the single initial state transformation, that is a subset $\oplus$ transformation on $Q_i = Q_0 = \{q_0, q_1\}$ and $q_i = q_0$.

So that $M' = (\{s, q_1, q_2, q_3\}, \{0, 1\}, \delta', Q_0', \{q_3\}, \oplus)$ where from definition 5.37

$$Q_0' \;=\; Q_0 \oplus Q_i \oplus \{s\} \;=\; \{s\},$$

and δ' can be given by (see also figure 5.18)

δ'	0	1
s	$\{s, q_1, q_2\}$	$\{q_2\}$
q_1	$\{s, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\{q_1, q_2\}$
q_3	$\{s, q_1\}$	$\{q_3, q_2\}$

.

[21] The $\oplus$-FA M in example 5.38 originates from example 5.35, page 121 where it is constructed from the minimal DFA. As the number of states in M is equal to minimal number of states in a reconstructed $\oplus$-FA, M is minimal.

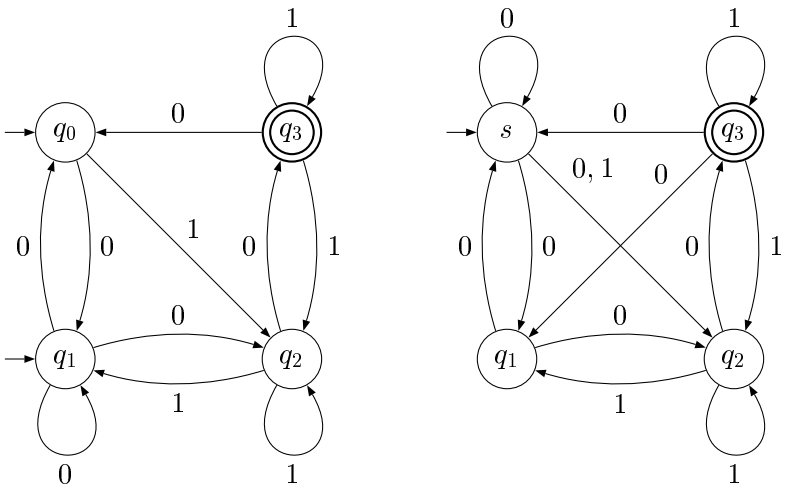

Figure 5.18: Example 5.41: Left M; Right M'

Theorem 5.42 *For every $\oplus$-FA, an equivalent $\oplus$-FA can be constructed with the same number of states and exactly one initial state.*

Proof Let M be a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$, we can construct $M' = (Q', \Sigma, \delta, Q_0', F', \oplus)$ using the subset $\oplus$ transformation on the subset $Q_i = Q_0$ and state $q_i \in Q_0$. Then $Q' = Q \oplus \{q_i\} \oplus \{s\}$ and

$$Q_0' \;=\; Q_0 \oplus Q_0 \oplus \{s\} \;=\; \{s\}.$$

Thus $|Q'| = |Q|$, $|Q_0'| = 1$ and $M' \equiv M$ so that for every $\oplus$-FA an equivalent $\oplus$-FA can be constructed with the same number of states and exactly one initial state. □

We use this result in two ways: the first is in the subset assignment process of subset reconstruction. If a $\oplus$-FA with a certain number of states exists which recognizes the language of a minimal DFA, then there exists an equivalent $\oplus$-FA with the same number of states but exactly one initial state. Thus when generating a subset assignment for a given minimal DFA for use in subset reconstruction we need only consider those subset assignments resulting in one initial state. Furthermore from the subset $\oplus$ transformation we can combine sets of states into a single state and thus can ignore large groups of equivalent subset assignments reducing the computational complexity of subset reconstruction.

Another form in which the single initial state transformation could be used is that of storage of the $\oplus$-FA. As we will discuss later we can automatically apply the single initial state transformation before storing a $\oplus$-FA and store this information implicitly as the first state stored.

The single initial state transformation and the format of $\oplus$-FA seems to imply that a similar transformation is available for the final state set.

However, this is not the case as the subset $\oplus$ transformation does not remove Q_i from it, but only the state replaced. If the state replaced was a final state then so is the constructed state, thus the number of final states remains constant.

Besides these applications of the subset $\oplus$ transformation, we can use the transformation to reduce the number of states and transitions in a $\oplus$-FA as shown in the next two sections.

Reduction of States

In this section we describe how the subset $\oplus$ transformation can be used to reduce the number of states in a $\oplus$-FA. We provide examples of the reduction and describe how these states may be detected and reduction algorithms based on the subset $\oplus$ transformation.

It is important to note that as the subset $\oplus$ transformation generates equivalent $\oplus$-FA with the same number of states, it does not reduce the number of states in a $\oplus$-FA. However, what it can do is allow for reducible states to be more easily detectable. Firstly we have simple algorithms to detect and remove initial and final-useless states[22], we plan to transform the reducible states into either of these types to be able to detect and remove them.

The main two targets of such transformations are the redundant and equivalent states. However, due to the nature of the $\oplus$ operator we have been unable to be able to easily distinguish reducible equivalent states to non-reducible equivalent states.

Before discussing how subset $\oplus$ transformations can be used in this regard we present two examples of $\oplus$-FA reduction through the use of subset $\oplus$ transformations.

Example 5.43

Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_0\}, \{q_3\}, \oplus)$ with δ given by (see also figure 5.19)

δ	0	1
q_0	$\{q_1, q_2\}$	$\{q_0\}$
q_1	$\{q_1\}$	$\{q_0, q_3\}$
q_2	$\{q_2\}$	$\{q_1, q_2\}$
q_3	$\{q_3\}$	$\{q_1, q_2\}$

.

We can construct $\oplus$-FA M' with $Q_i = \{q_1, q_3\}$ and $q_i = q_1$. So that $M' = (\{q_0, s, q_2, q_3\}, \{0, 1\}, \delta', \{q_0\}, \{q_2\}, \oplus)$ where from definition 5.37 and

[22] See section 5.2.1.

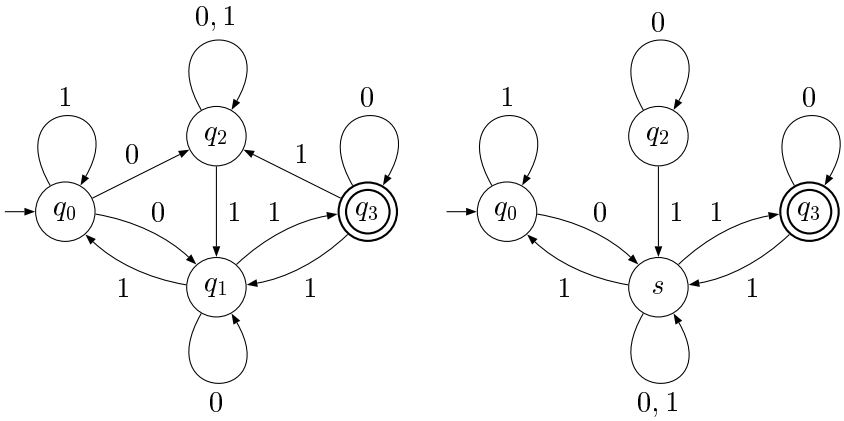

Figure 5.19: Example 5.43: Left M; Right M'

δ' can be given by (see also figure 5.19)

δ'	0	1
q_0	$\{s\}$	$\{q_0\}$
s	$\{s\}$	$\{q_0, s, q_3\}$
q_2	$\{q_2\}$	$\{s\}$
q_3	$\{q_3\}$	$\{s\}$

.

In M' the state q_2 is initial-useless and can be removed. This means that whenever the state q_2 is set active in M, the state q_1 is set active as well, furthermore if the state q_1 in set active in M while q_2 is inactive, the states set in M' would contain the subset $\{s, q_2\}$, as q_2 is initial-useless this is impossible and thus q_1 is active only when q_2 is active in M and thus the q_1 is redundant to q_2 in M.

Example 5.44

Let M be a $\oplus$-FA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_0\}, \{q_3\}, \oplus)$ with δ given by (see also figure 5.19)

δ	0	1
q_0	$\{q_1\}$	$\{q_0, q_3\}$
q_1	$\{q_1\}$	$\{q_3\}$
q_2	$\{q_1\}$	$\{q_3\}$
q_3	$\{q_0, q_3\}$	$\{q_1, q_2\}$

.

We can construct $\oplus$-FA M' with $Q_i = \{q_1, q_3\}$ and $q_i = q_1$. So that $M' = (\{q_0, s, q_2, q_3\}, \{0, 1\}, \delta', \{q_0\}, \{q_2\}, \oplus)$ where from definition 5.37 and

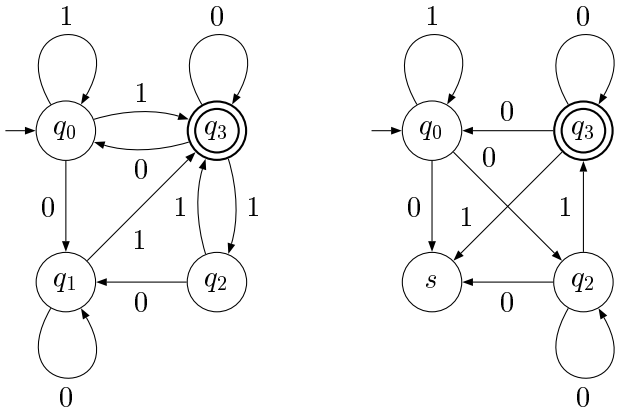

Figure 5.20: Example 5.44: Left M; Right M'

δ' can be given by (see also figure 5.20)

δ'	0	1
q_0	$\{s, q_2\}$	$\{q_0\}$
s	$\emptyset$	$\emptyset$
q_2	$\{s, q_2\}$	$\{q_3\}$
q_3	$\{q_0, q_3\}$	$\{s\}$

.

In M' the state s is final-useless and can be removed. It is of interest to note that the states q_1 and q_2 share all their transitions and are such equivalent. Although not all equivalent states in a $\oplus$-FA are reducible, equivalent states of this format are reducible as through subset $\oplus$ construction.

Examples 5.43 and 5.44 reduce $\oplus$-FAs by transforming redundant and equivalent states into initial and final-useless states respectively.

We now show that these states are always detectable through the use of the subset $\oplus$ transformation as well as give the limitation of the equivalent states.

Theorem 5.45 *In a $\oplus$-FA redundant states are detectable as initial-useless states through the subset $\oplus$ transformation.*

Proof Let M be a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with the states $q_i, q_j \in Q$ redundant, so that $pr(q_i) = pr(q_j)$. By constructing the $\oplus$-FA $M' = (Q', \Sigma, \delta', Q_0', F', \oplus)$ through the subset $\oplus$ transformation on the set $\{q_i, q_j\}$ by replacing the state q_i with a state s representing the set $\{q_i, q_j\}$. We must show that the state q_j is initial-useless in M'.

If q_j was not initial useless in M' then there exists a word $x \in \Sigma^*$ so that $q_j \in \delta'(Q_0', x)$. There are two cases where this can happen, namely where $s \in \delta'(Q_0', x)$ and $s \notin \delta'(Q_0', x)$.

If $s \in \delta'(Q_0', x)$ then as $s = \{q_i, q_j\}$, $q_i \in \delta(Q_0, x)$ and $q_j \notin \delta(Q_0, x)$ but as $pr(q_i) = pr(q_j)$ this is impossible.

If $s \notin \delta'(Q_0', x)$ then as $s = \{q_i, q_j\}$, $q_i \notin \delta(Q_0, x)$ and $q_j \in \delta(Q_0, x)$ but as $pr(q_j) = pr(q_i)$ this is impossible.

So that redundant states in a $\oplus$-FA are detectable as initial-useless through the subset $\oplus$ transformation. □

Theorem 5.46 *In a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with the states $q_i, q_j \in Q$ so that for every word $x \in \Sigma^*$ so that $\delta(\{q_i, q_j\}, x) \cap F = \emptyset$, are detectable as final-useless states through the subset $\oplus$ transformation.*

Proof Let M be a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with the states $q_i, q_j \in Q$ so that for every word $x \in \Sigma^*$ so that $\delta(\{q_i, q_j\}, x) \cap F = \emptyset$. By constructing the $\oplus$-FA $M' = (Q', \Sigma, \delta', Q_0', F', \oplus)$ through the subset $\oplus$ transformation on the set $\{q_i, q_j\}$ by replacing the state q_i with a state s representing the set $\{q_i, q_j\}$. We must show that the state s is final-useless in M'.

Note that as $\delta(\{q_i, q_j\}, \epsilon) \cap F = \emptyset$ the states $q_i, q_j \notin F$ so that $F = F'$.

If s was not final useless in M' then there exists a word $x \in \Sigma^*$ so that $\delta'(s, x) \cap F' \neq \emptyset$, but this implies that $\delta(q_i, q_j), x) \cap F \neq \emptyset$. This forms a contradiction to $\delta(q_i, q_j), x) \cap F = \emptyset$ so that s is final-useless.

Thus states where $\delta(\{q_i, q_j\}, x) \cap F = \emptyset$ for all x are detectable as final-useless through the subset $\oplus$ transformation. □

As discussed in chapter 5, equivalent states are not always reducible in a $\oplus$-FA. However, as the above theorem shows a subset of equivalent states, that is those state sets which are final-useless in a $\oplus$-FA, can be transformed into a single final-useless state through the subset $\oplus$ transformation. These subset of equivalent states include those with identical transitions as in example 5.44 but also those states whose transitions interact with each other in such a way that final states are unreachable. This second type of equivalent state is not as easily detectable through normal algorithm and as such the subset $\oplus$ transformation is useful in such a circumstance.

By using the subset $\oplus$ transformations as described above we can remove all redundant and equivalent states of the type described in theorem 5.46. Thus an algorithm for reduction of a $\oplus$-FA could be to check every pair of states through subset $\oplus$ transformation for generated initial-useless and/or final-useless. In a $\oplus$-FA M with n states there would be $n^2 - n$ such subsets. As the subset $\oplus$ transformation has computational complexity of $\mathcal{O}(m)$ with m the number of states, and excluding the algorithm for initial-useless and final-useless states such a reduction algorithm would have at most computational complexity of $\mathcal{O}(mn^2)$ where m is the number of transitions and n the number of states.

It is important to note in such an implementation that the subset $\oplus$ transformation does not detect all reducible states and as such the resultant

$\oplus$-FA is not always minimal. Also only sets of size 2 are checked for redundancy and equivalence. This does not mean that larger set of equivalent or reducible states will not be detected and removed. As each state in a redundant state set is redundant with every other state in the set, the entire set will be detected and removed a pair at a time. Equivalent state sets which are final-useless consisting of more than two states can be detected by combining the states in the set until exactly two states remain. The reduction algorithm can be modified to include this case by continuously checking the resultant $\oplus$-FAs for reducible states until a certain threshold is reached.

This concludes the reduction of states through the use of the subset $\oplus$ transformation. The final application of the subset $\oplus$ transformation we will discuss is the property of the reduction of transitions.

Reduction of Transitions

As with the NFA reduction algorithms we focus mainly on the reduction of states for $\oplus$-FAs, although the reduction of transitions is also important to consider when the storage of the $\oplus$-FA is to be considered. The subset $\oplus$ transformation presents not only an algorithm to detect and reduce the number of states but also the number of transitions in the $\oplus$-FA.

The subset $\oplus$ transformation replaces any transitions to the removed state to the constructed state and the states in the set the constructed state represents. Similarly the constructed states have the combined transitions of all the transitions of the set it represents. Initially this may seem counter productive as we wish to reduce the number of transitions in the $\oplus$-FA. However, as the transitions are combined using the $\oplus$ operator, the resultant number of transition may decrease.

When replacing a state q_i in the $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with a state s representing the set Q_i where $Q_i \subseteq Q$ and $q_i \in Q$ the number of transitions will decrease if the following conditions hold:

> The number of transitions added are less than the number of transitions removed through the $\oplus$ operator,

By limiting the state set size to two states, that is $Q_i - \{q_i, q_j\}$, we have that the number of transitions are reduced if

$$\sum_{\sigma \in \Sigma} \left(|\delta(q_i, \sigma)| - |\delta(Q_i, \sigma)| \; - \; \sum_{q} \left(|Q_i| - |\delta(q, \sigma) \cap Q_i| \right) \right) > 0,$$

with $q \in Q$ so that $q_i \in \delta(q, \sigma)$.

Where the formula represents the sum over all alphabet symbols of the transitions removed from the state q_i less the transitions added from the state s less the difference of transitions resulting from the transitions to the state s and the set $Q_i - \{q_i\}$.

Although determining these conditions using a search through the possible combinations may seem to be quite time consuming, we can attempt the subset $\oplus$ transformation without finding these conditions on every set of states, similar to the reduction process of the previous section, and keep only the transformations which reduce the number of states.

We show an example of the reduction of states through the use of subset $\oplus$ transformation before describing circumstances in which the reduction of transitions is probable to occur.

Example 5.47 (Transition Reduction)
Let M be a $\oplus$-FA where $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_0\}, \{q_3\}, \oplus)$ with δ given by (see also figure 5.21)

δ	0	1
q_0	$\{q_0, q_1\}$	$\{q_1, q_2\}$
q_1	$\{q_0, q_2, q_3\}$	$\{q_2, q_3\}$
q_2	$\{q_0, q_3\}$	$\{q_2\}$
q_3	$\{q_1, q_2\}$	$\{q_2, q_3\}$

.

By replacing the state q_1 with the state s representing the subset $\{q_1, q_2\}$ results in the $\oplus$-FA $M' = (\{q_0, s, q_2, q_3\}, \{0, 1\}, \delta', \{q_0\}, \{q_3\}, \oplus)$ with δ' given by (see also figure 5.21)

δ	0	1
q_0	$\{q_0, s, q_2\}$	$\{s\}$
s	$\{q_2\}$	$\{q_3\}$
q_2	$\{q_0, q_3\}$	$\{q_2\}$
q_3	$\{s\}$	$\{q_2, q_3\}$

.

The $\oplus$-FA M has 16 transitions while M' contains 12 transitions, resulting in a reduction of 4 transitions.

This concludes the minimization and reduction of $\oplus$-FA. We have discussed an algorithm to generate a minimal unary $\oplus$-FA with one final state based on LFSRs which given the final state sequence performs in reasonable time. While this algorithm could be modified to generate a minimal unary $\oplus$-FA with more than one final state, the computational complexity increases dramatically.

We began our discussion of reduction and minimization of non-unary $\oplus$-FA by examining the reducible states for NFAs in the context of $\oplus$-FAs. Although the initial-useless, final-useless and redundant states remained reducible, not all of the equivalent and contained states were reducible. This factor prohibits many of the NFA minimization ideas to function. The first algorithm we discussed was that of subset reconstruction. This was a modification of Kameda and Weiner's NFA minimization algorithm [11]. We then presented the adapted subset reconstruction operation for $\oplus$-FAs. This

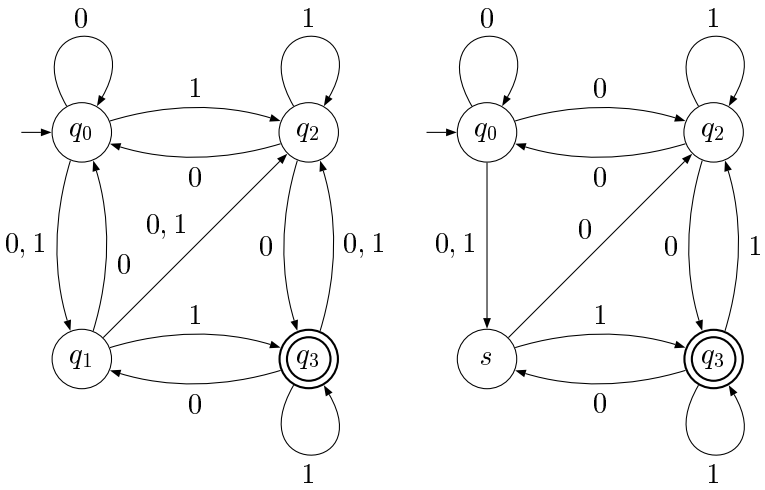

Figure 5.21: Example 5.47: Left M; Right M'

algorithm guaranteed a minimal $\oplus$-FA. However, due to the problem of constructing a dual for $\oplus$-FAs we were unable to present an algorithm for finding a suitable subset assignment short of a brute force search.

The subset symmetric difference transformation simplified this process as it presented an algorithm to construct an equivalent $\oplus$-FA with the same number of states. This reduces the possible subset assignments for the subset reconstruction. Furthermore this transformation presented an algorithm to construct an equivalent $\oplus$-FA with exactly one initial state, as well as reducing both states and transitions.

Before continuing onto the next chapter we present a section on the adaptation of the generalized subset reconstruction operation from this chapter adapted for specific use on $\cap$-FA. The reason for this is to show that the generalized algorithm we presented can be adapted to other forms of $*$-FA, as well as providing some insight into the reduction algorithms of other forms of $*$-FA.

5.3 Subset Reconstruction - $\cap$-FA

In section 5.2.3 (page 117) we presented a generalized form of the intersection rule[23], namely the subset reconstruction operation[24]. We then adapted this generalized subset reconstruction operation for use on the $\oplus$-FA. In this section we discuss the specific adaptation of this generalized rule for use with $\cap$-FAs.

The intersection rule for the reconstruction of NFAs was constructed taking advantage of the following property of DFAs constructed through subset construction.

[23]See definition 4.24, page 58.
[24]See definition 5.32, page 117.

Let M be a NFA $M = (Q, \Sigma, \delta, Q_0, F)$ and let M' be the equivalent DFA through subset construction with $M' = \text{DFA}(M) = (P, \Sigma, \delta', p_0, F')$. If $q \in p$ where $q \in Q$ and $p \in P$ then for every $\sigma \in \Sigma$,

$$\delta'(p, \sigma) = \bigcup_{q' \in p} \delta(q', \sigma).$$

so that $\delta(q, \sigma) \subseteq \delta'(p, \sigma)$.

This means that if P_q is the set of all state $p \in P$ so that $q \in p$ then

$$\delta(q, \sigma) \subseteq \bigcap_{p \in P_q} \delta(p, \sigma).$$

Similarly if M is a $\cap$-FA, with $M = (Q, \Sigma, \delta, Q_0, F, \cap)$ and M' the equivalent DFA through subset construction with $M' = \text{DFA}(M) = (P, \Sigma, \delta', p_0, F')$, we have that if $q \in p$ where $q \in Q$ and $p \in P$ then for every $\sigma \in \Sigma$

$$\delta'(p, \sigma) = \bigcap_{q' \in p} \delta(q', \sigma),$$

so that $\delta'(p, \sigma) \subseteq \delta(q, \sigma)$.

This means that with the set P_q as described above,

$$\delta(q, \sigma) \subseteq \bigcup_{p \in P_q} \delta(p, \sigma).$$

Thus instead of using the intersection of the transitions of the states in the DFA containing a specific state we take the union as shown in the following example:

Example 5.48 ($\cap$-FA Subset Reconstruction)
Let M' be the minimal DFA $M' = (\{p_0, p_1, \ldots, p_7\}, \{0, 1\}, \delta', p_0, \{p_6, p_7\})$ with δ' given by (see also figure 5.22)

δ'	0	1
p_0	p_1	p_2
p_1	p_0	$\emptyset$
p_2	p_3	p_4
p_3	p_2	p_5
p_4	$\emptyset$	p_6
p_5	p_2	$\emptyset$
p_6	p_4	p_7
p_7	p_4	$\emptyset$

.

To construct a $\cap$-FA $M = (Q, \Sigma, \delta, Q_0, F, \cap)$ from M' we first need a subset assignment which we define as follows: $< Q, f >$ where $Q =$

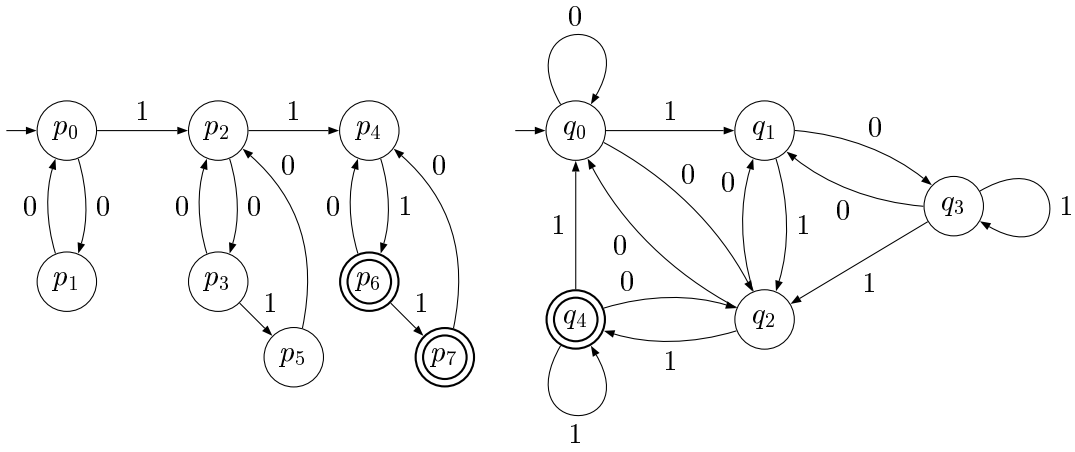

Figure 5.22: Example 5.48: Left M'; Right M

$\{q_0, q_1, q_2, q_3, q_4\}$ *and* f *is described as follows:*

p	$f(p)$
p_0	$\{q_0\}$
p_1	$\{q_0, q_2\}$
p_2	$\{q_1\}$
p_3	$\{q_3\}$
p_4	$\{q_2\}$
p_5	$\{q_2, q_3\}$
p_6	$\{q_4\}$
p_7	$\{q_0, q_4\}$.

Clearly $F = \{q_4\}$*, and* $Q_0 = \{q_0\}$*. Also,* δ *is constructed as follows (the resultant transitions are shown in figure 5.22):*

$$
\begin{aligned}
\delta(q_0,0) &= \delta(p_0,0) \cup \delta(p_1,0) \cup \delta(p_7,0) = p_1 \cup p_0 \cup p_4 = \{q_0, q_2\} \\
\delta(q_0,1) &= \delta(p_0,1) \cup \delta(p_1,1) \cup \delta(p_7,1) = p_2 \cup \emptyset \cup \emptyset = \{q_1\} \\
\delta(q_1,0) &= \delta(p_2,0) = p_3 = \{q_3\} \\
\delta(q_1,1) &= \delta(p_2,1) = p_4 = \{q_2\} \\
\delta(q_2,0) &= \delta(p_1,0) \cup \delta(p_4,0) \cup \delta(p_5,0) = p_0 \cup \emptyset \cup p_2 = \{q_0, q_1\} \\
\delta(q_2,1) &= \delta(p_1,1) \cup \delta(p_4,1) \cup \delta(p_5,1) = \emptyset \cup p_6 \cup \emptyset = \{q_4\} \\
\delta(q_3,0) &= \delta(p_3,0) \cup \delta(p_5,0) = p_2 \cup p_2 = \{q_1\} \\
\delta(q_3,1) &= \delta(p_3,1) \cup \delta(p_5,1) = p_5 \cup \emptyset = \{q_2, q_3\} \\
\delta(q_4,0) &= \delta(p_6,0) \cup \delta(p_7,0) = p_4 \cup p_4 = \{q_2\} \\
\delta(q_4,1) &= \delta(p_6,1) \cup \delta(p_7,1) = p_7 \cup \emptyset = \{q_0, q_4\}.
\end{aligned}
$$

It can be confirmed that DFA(M) is isomorphic to M' so that $\mathcal{L}(M) = \mathcal{L}(M')$ so that we have constructed a $\cap$-FA from a given DFA through subset reconstruction. In the example when calculating $\delta(q_0, \sigma)$ we do not necessarily have to calculate $\delta'(p_0, \sigma) \cap \delta'(p_1, \sigma) \cap \delta'(p_7, \sigma)$ as we have that $f(p_0) = \{q_0\}$ so that $\delta(q_0, \sigma) = \delta'(p_0, \sigma)$. This is true for all mappings from a state in the DFA to a single state in the $*$-FA, this can also be used to

calculate discrepancies in the reconstructed $*$-FA by comparing the states resultant transitions to its mappings transitions.

We can now formally present the subset reconstruction for $\cap$-FA from a given DFA as follows:

Definition 5.49 (Subset Reconstruction: $\cap$-FA)
Let M be a DFA $M' = (P, \Sigma, \delta', p_0, F')$ and a subset assignment $< Q, f >$ we can construct a $\cap$-FA $M = (Q, \Sigma, \delta, Q_0, F, \cap)$ as follows:

- $Q_0 \leftarrow f(p_0)$.
- $q \in F$ *if* $\forall p \in P$ *so that* $q \in f(p)$, $p \in F'$, *and*
- *For* $q \in Q$ *and* $\sigma \in \Sigma$

$$\delta(q, \sigma) \;=\; \bigcup_{q \in f(p)} f(\delta'(p, \sigma)),$$

or if $\exists p \in P$ *so that* $f(p) = \{q\}$ *then*

$$\delta(q, \sigma) \;=\; f(\delta'(p, \sigma)),$$

This concludes our discussion on the adaptation of the generalized algorithms for $*$-FAs with regards to $\cap$-FA.

In the next chapter we discuss the compression of $\oplus$-FAs, that is, the storage of $\oplus$-FAs both in terms of time and space in the application of such $\oplus$-FAs.

Chapter 6

Compression

In the previous chapter we discussed the reduction and minimization of $\oplus$-FA. However, if these reduced $\oplus$-FAs are to be used successfully in a practical application, then we would require an efficient method in which to access and run the $\oplus$-FAs. So, although this thesis focuses on the reduction and minimization of $\oplus$-FAs, in this chapter we step away from the reduction processes and instead discuss storage formats for $\oplus$-FAs.

For practical applications we could simply construct a storage format which dumps the various characteristics of the $\oplus$-FA into a binary file. However, we are interested in efficient compressed storage formats for efficiency. That is, we wish to construct a storage format consisting of as few bits as possible.

A compressed storage format may seem an excessive step if we have reduction techniques for $\oplus$-FAs which reduce the amount of data which needs to be stored. However, even minimal $\oplus$-FAs may consist of millions of states in which case the memory requirements of the $\oplus$-FA become an inhibitant factor for use of $\oplus$-FAs.

In this chapter we discuss two distinct storage formats. The first storage format is constructed for use with unary $\oplus$-FAs based on the compression algorithm from section 5.1. The second storage format is for use with the general $\oplus$-FAs based on a storage algorithm for dictionaries from [4].

When discussing the storage formats we will focus on two main attributes of these formats. Namely, the compression and execution of the formats. The compression attribute deals with the size of the stored binary sequence. By execution we mean the process that is used to transform the stored data into a useable format and execute the stored $\oplus$-FA.

We wish to construct a storage format which has both a good compression ratio and whose computational complexity for execution is small. With this in mind we begin our discussion of a storage format for unary $\oplus$-FAs.

6.1 Unary $\oplus$-FA

As stated above, we start our discussion with the simplified case of unary $\oplus$-FAs. The storage format is based on the unary $\oplus$-FA minimization algorithm from section 5.1.

The basis of this format is from the third step of the minimization algorithm (see page 85) where the minimal unary $\oplus$-FA with one final state is constructed. We discuss the storage format by first examining the minimization algorithm. After that, we provide a formal format for the unary $\oplus$-FA and show how this format can be used to run a unary $\oplus$-FA efficiently.

In the construction process, we need to be able to construct the $\oplus$-FA from a sequence of stored bits. This means that the five elements of a $\oplus$-FA need to be represented as a sequence of bits. These elements are: the start state set, Q; the alphabet, Σ; the initial state set, Q_0; the final state set, F; and the transition function, δ.

These elements can either be stored *explicitly* or *implicitly*. An explicitly stored element is read directly from the binary sequence without need of excessive computation to calculate the element. An implicitly stored element is stored in such a way that it needs to be derived from the stored elements.

It is important to note that we do not make use of standard compression techniques, such as pattern compression, in these storage formats. We do not include such techniques as we wish to create a general storage format which can be used a basis for application specific storage formats. Instead the storage formats are constructed based only on the properties of the $\oplus$-FAs.

Recall that the algorithm from section 5.1 is divided into three steps. The first step constructs the final state sequence which represents the language accepted by the unary $\oplus$-FA. The second step constructs the characteristic polynomial of minimal degree which generated the final state sequence. The third and final step uses the final state sequence as well as the characteristic polynomial to construct the minimal unary $\oplus$-FA with one final state.

It is this third step that we are interested in, as it uses two elements of the unary $\oplus$-FA to reconstruct an equivalent unary $\oplus$-FA with one final state.

6.1.1 Preliminaries

The third step of the minimization algorithm from section 5.1 constructs a minimal unary $\oplus$-FA with one final state from the minimal characteristic polynomial and final state sequence. Instead of explicitly storing the five elements of the $\oplus$-FA, namely the state set, alphabet, transition function, initial and final state sets, we could instead store only these two elements and reconstruct the unary $\oplus$-FA via step 3 of the algorithm. However, storing a

unary $\oplus$-FA in such a format entails some computation to generate the five elements of the $\oplus$-FA.

In an attempt to remove this initial computation from the storage format, we attempt to find an alternative compressed storage format by examining the reconstruction step of the algorithm. Our goal is to determine the minimum amount of information required to calculate the five elements of the unary $\oplus$-FA from the minimization algorithm while keeping the reconstruction process simple.

We start by examining the language of a unary $\oplus$-FA. As a unary $\oplus$-FA has exactly one alphabet symbol, we can assume the alphabet symbol to be a. As such it need not be stored[1].

We are now left with four elements which need to be determined. At this stage of the algorithm we are given the minimal characteristic polynomial and final state sequence of the unary $\oplus$-FA. Initially we use only the characteristic polynomial $c(X) = c_0 + c_1X^1 + \ldots + c_{n-1}X^{n-1}$ and determine which properties can be constructed from it.

As discussed in section 5.1 (page 77), we know that the number of states in the $\oplus$-FA is one greater than the order of the characteristic polynomial. That is, $|Q| = n$ where $n-1$ is the order of the characteristic polynomial, so that $Q = \{q_0, q_1, \ldots, q_{n-1}\}$. Furthermore, from the algorithm, there is only one final state which is assumed to be q_{n-1}. Lastly, the transition function can be determined by $c(X)$ by using the normal form of the characteristic matrix A which is defined as follows:

$$A = \begin{bmatrix} 0 & 0 & \ldots & 0 & c_0 \\ 1 & 0 & \ldots & 0 & c_1 \\ 0 & 1 & \ldots & 0 & c_2 \\ \vdots & \vdots & \ldots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & c_{n-2} \\ 0 & 0 & \ldots & 1 & c_{n-1} \end{bmatrix}.$$

This corresponds to each non-final state q_i, where $i \neq n-1$, having exactly one outgoing transition, namely $\delta(q_i, a) = \{q_{i+1}\}$. The final state q_{n-1} has a transition to any state q_i, with $i \leq n-1$ if and only if $c_i = 1$ in the characteristic polynomial[2].

To summarize from the characteristic polynomial $c(X)$ we can determine the state set, final state and transition function which leaves only the set of initial states to be determined.

Note 15 *When constructing the transition function we need not examine the characteristic polynomial to construct the transitions from the non-final*

[1] In the situation where the alphabet character should be defined it can be stored once in the initial byte of the storage format without affecting the storage format as a whole.

[2] These properties come directly from the algorithm from section 5.1.

states, as only the transitions of the final state are variable in the normal form of the characteristic matrix.

In the minimization algorithm the set of initial states is calculated using the normal form of the transition function and the first n bits of the final state sequence (page 84). It follows that, by storing the first n bits of the final state sequence in the storage format we can reconstruct the initial state set. We note that we would require n bits to store the final state sequence. Alternatively, we can store a binary sequence of the same length which represents the initial state set. By storing the initial state set we do not have the computation involved in calculating the initial state set as in the algorithm. Each bit of the initial state set sequence represents whether a state is an initial state or not. That is, for the states $\{q_0, q_1, \ldots, q_{n-1}\}$ and the binary sequence $s_0 s_1 \ldots s_{n-1}$, a state $q_i \in Q_0$ if and only if $s_i = 1$ where $i = 0, 1, \ldots, n-1$.

We can now formally define and propose a format to store the sequence of initial states and characteristic polynomial.

6.1.2 Storage Format

In the previous section we examined the unary $\oplus$-FA minimization algorithm and found that we could store the unary $\oplus$-FA with one final state. In this section we formally define this format and discuss factors influencing the storage format as well as providing examples of the format.

We define this format, followed by an example, as follows:

Format 6.1

Given a minimal unary $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with one final state constructed from the characteristic polynomial

$$c(X) = c_{n-1}X^{n-1} + \ldots + c_1 X^1 + c_0$$

and final state sequence f as in section 5.1, M can be stored as follows[3]*:*

$$\langle\ (Q_0)_2\ ,\ (c(X))_2\ \rangle,$$

where $(Q_0)_2$ is a binary sequence representation of the initial state set, and $(c(X))_2$ is a binary sequence representation of the characteristic polynomial.

These sequences are defined as follows:

- *$(Q_0)_2$ is a bit sequence $i_0 i_1 \ldots i_{n-1}$, where $i_j = 1$ if and only if $q_j \in Q_0$, and*

[3]In this section we use the following notations for storage formats. The symbols " $<$ " and " $>$ " denote a storage format. The notation "$(x)_2$" represents a binary sequence and the comma symbol separates logical sequences from each other.

- $(c(X))_2$ *is a bit sequence* $c_0 c_1 \ldots c_{n-1}$*, as defined for the characteristic polynomial.*[4]

Example 6.2 (Unary Storage)

Let M' *be the unary* $\oplus$*-FA with one final state from example 5.17 (page 91). We have that* $M' = (\{q'_0, q'_1, q'_2, q'_3, q'_4, q'_5\}, \{a\}, \delta', \{q'_0, q'_2, q'_4\}, \{q'_5\}, \oplus)$ *where* δ' *is given by (also shown in figure 6.1 left)*

δ'	a
q'_0	$\{q'_1\}$
q'_1	$\{q'_2\}$
q'_2	$\{q'_3\}$
q'_3	$\{q'_4\}$
q'_4	$\{q'_5\}$
q'_5	$\{q'_0, q'_1, q'_2, q'_3, q'_4, q'_5\}$

.

Furthermore M' *has the characteristic function* $c(X) = 1 + X^1 + X^2 + X^3 + X^4 + X^5$ *and final state sequence* $f = 0111111$ *with a cycle of length five.*

We store M' *as the binary sequence* $\langle (Q_0)_2,\ c(X) \rangle$*, which is encoded as follows:*

- $(Q_0)_2 = 101010$ *representing the initial state set* $\{q'_0, q'_2, q'_4\}$*, and*
- $c(X) = 111111$ *representing the characteristic polynomial* $1 + X^1 + X^2 + X^3 + X^4 + X^5 + X^6$*.*

This results in the binary sequence 101010111111 *consisting of 12 bits.*

In the above example we generated a binary sequence in the storage format for a unary $\oplus$-FA with one final state. For use in applications there needs to be a method to be able to reconstruct the original unary $\oplus$-FA with one final state from the binary sequence. We now provide the description of this reconstruction, after which we provide an example.

A unary $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with one final state can be constructed from the stored binary sequence $\langle (Q_0)_2,\ (c(X))_2 \rangle$ as follows:

- As we are dealing with unary $\oplus$-FA, we have that the alphabet $\Sigma = \{a\}$.
- To construct the state set we look at the length of the binary sequence $\langle (Q_0)_2,\ (c(X))_2 \rangle$. As the subsequences have equal length, n bits, and each bit in the subsequence corresponds to one state we can construct the state set $Q = \{q_0, q_1, \ldots, q_{n-1}\}$.

[4] Recall that $c_i \in \{0, 1\}$ for $0 \leq i \leq n-1$, because $c(X)$ is a polynomial in GF(2).

- Next we construct the final state set. From the minimization algorithm we assume the single final state to be the last state, so that $F = \{q_{n-1}\}$.
- The initial state set Q_0 can be constructed from $(Q_0)_2$ as follows:

$$q_j \in Q_0 \;\Leftrightarrow\; i_j = 1, \text{ with } 0 \leq j \leq n-1$$

- The transition function δ can be constructed from of the characteristic polynomial $c(X)$ as follows:

$$\begin{array}{rcl} \delta(q_i, a) & = & q_{i+1},\ 0 \leq i \leq n-1 \\ \delta(q_{n-1}, a) & = & \{q_i |\ c_i = 1\}. \end{array}$$

Example 6.3 (Unary Reconstruction)

Given the binary sequence 101010111111 *from example 6.2 (page 145), we wish to construct a minimal unary $\oplus$-FA with one final state equivalent to M from example 6.2.*

From the format we have that $\langle (Q_0)_2\ ,\ (c(X))_2 \rangle = 101010111111$ *with* $(Q_0)_2$ *and* $(c(X))_2$ *of equal length so that:*

$$\begin{array}{rcl} (Q_0)_2 & = & 101010, \\ (c(X))_2 & = & 111111. \end{array}$$

We will construct the unary $\oplus$-FA $M' = (Q', \Sigma, \delta', Q_0', F', \oplus)$ *as described on page 145 as follows:*

- *We have that* $\Sigma = \{a\}$.
- *As* $|(Q_0)_2| = 6$ *we that* $Q' = \{q_0', q_1', q_2', q_3', q_4', q_5'\}$.
- *We have that* $F' = \{q_{n-1}'\} = \{q_5'\}$.
- *We have that* $(Q_0)_2 = 101010$ *so that* $Q_0' = \{q_0', q_2', q_4'\}$.
- *Lastly we reconstruct the transition function from the normal form of the transition matrix as follows:*

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & c_0 \\ 1 & 0 & 0 & 0 & 0 & c_1 \\ 0 & 1 & 0 & 0 & 0 & c_2 \\ 0 & 0 & 1 & 0 & 0 & c_3 \\ 0 & 0 & 0 & 1 & 0 & c_4 \\ 0 & 0 & 0 & 0 & 1 & c_5 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

The resultant unary $\oplus$-FA with one final state is shown in figure 6.1 right. As shown in the image the reconstructed M' is identical to the original M.

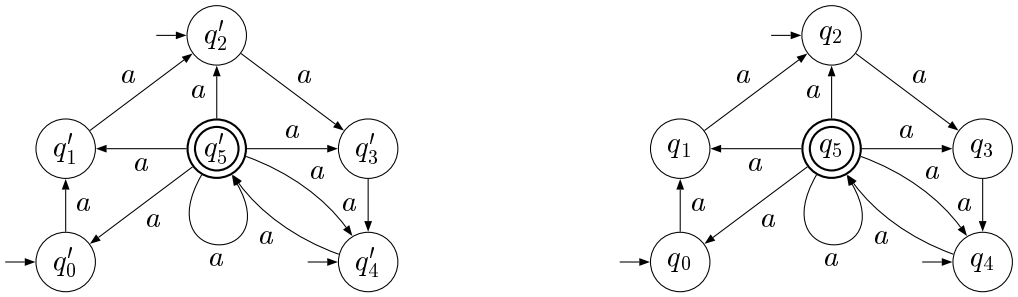

Figure 6.1: Left: Example 6.2: M'; Right: Example 6.3: M

We have discussed and provided an example of both the construction and reconstruction of unary $\oplus$-FAs with the storage format. Before discussing the use of the storage format in an application we discuss one issue regarding the size of the storage format.

The resultant size of a unary $\oplus$-FA stored in this format is $2n$ bits. However, as we are dealing with the storage of the unary $\oplus$-FA onto a computer, the format needs to fit into bytes of data. That is the length of the format in bits must be divisible by 8. An initial attempt to do this would be to zero-pad the end of the stored data. The issue with this is how to know whether the zeros at the end of the format are part of the sequence or the zero pad.

Instead we pad both the initial state binary sequence and characteristic polynomial sequence with 0s as follows:

$$\langle p,\ (Q_0)_2,\ p,\ (c(X))_2 \rangle\,,$$

where $p = 0 \ldots 0$ and $|p| = 8 - n \pmod 8$.

For example if $n = 14$ then $|p| = 8 - 14 = 8 - 6 \equiv 2 \pmod 8$ so that $p = 00$.

The padded 0's represented by p can be detected and removed from the stored $\oplus$-FA as shown below.

Given a binary sequence $\langle (Q'_0)_2,\ (c'(X))_2 \rangle = \langle p,\ (Q_0)_2,\ p,\ (c(X))_2 \rangle$ where

$$(Q'_0)_2 = i_{-m+1} \ldots i_0 i_1 \ldots i_{n-1} \text{ and } (c'(X))_2 = c_{-m+1} \ldots c_0 c_1 \ldots c_{n-1}$$

with

$$c_{-m+1} = i_{-m+1} = c_{-m+2} = i_{-m+2} = \ldots = c_{-1} = i_{-1} = 0$$

we have that:

- $\Sigma = \{a\}$,

- $Q = \{q_{-m+1} \ldots q_0, q_1, \ldots q_{n-1}\}$,
- $F = \{q_{n-1}\}$,
- Q_0 is constructed from $(Q_0')_2$ as follows:

$$q_j \in Q_0 \;\Leftrightarrow\; i_j = 1, \text{ and}$$

- δ is constructed from the characteristic polynomial as follows:

$$\begin{bmatrix} 0 & \ldots & 0 & 0 & \ldots & 0 & 0 \\ 1 & \ldots & 0 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ldots & \vdots & \vdots \\ 0 & \ldots & 1 & 0 & \ldots & 0 & 0 \\ 0 & \ldots & 0 & 1 & \ldots & 0 & c_0 \\ \vdots & \vdots & \vdots & \vdots & \ldots & \vdots & \vdots \\ 0 & \ldots & 0 & 0 & \ldots & 0 & c_{n-2} \\ 0 & \ldots & 0 & 0 & \ldots & 1 & c_{n-1} \end{bmatrix}.$$

The constructed unary $\oplus$-FA consists of more states than the original unary $\oplus$-FA. However, the states $q_{-m+1}, \ldots, q_{-1}$ can be identified as initial useless and reduced as follows:

- There exists no state q_j so that $q_{-m+1} \in \delta(q_j, a)$ and $q_{-m+1} \notin Q_0$. This means that q_{-m+1} is unreachable or initial-useless and so can be removed.
- As q_{-m+1} is initial useless and no state q_j with $j \neq -m+1$ exists so that $q_{-m+2} \in \delta(q_j, a)$ and $q_{-m+2} \notin Q_0$, q_{-m+2} can be removed.
- Similarly, the states $q_{-m+3}, \ldots, q_{-1}$ can be shown to be initial useless.

Thus when constructing the unary $\oplus$-FA from the sequence

$$\langle p \ , \ (Q_0)_2 \ , \ p \ , \ (c(X))_2 \rangle \, ,$$

we can detect and remove the initial-useless states in the storage format by examining the two subsequences p , $(Q_0)_2$ and p , $(c(X))_2$. As the first $|p|$ bits of the two sequences are 0s the states they would represent are initial-useless, as described above, and can be reduced when constructing the unary $\oplus$-FA.

This process can be simplified by calculating the length of p by examining the bits of the two subsequences until the first 1 is found. This is derived from the property that for every pair of bits, i_j and c_j, processed there can be four possiblities:

1. If $i_j = c_j = 0$ then the state is initial-useless as discussed above.

2. If $i_j = 1$ and $c_j = 0$ then the state is an element of the initial state set and is thus not initial-useless and cannot be removed.

3. If $i_j = 0$ and $c_j = 1$ then there exists a transition from the final state to this state and thus is not initial-useless.

4. If $i_j = c_j = 1$ then the state is an element of the initial state set and there exists a transition from the final state to this state and thus is not initial-useless.

We now provide an example of the padded storage format after which we discuss how the storage format could be used effectively in an application.

Example 6.4 (Unary Storage)
Let M be the unary $\oplus$-FA with one final state from example 6.2 (page 145). From the example we have that the unpadded stored binary sequence is

$$101010111111,$$

of length 12 where $|(Q_0)_2| = |(c(X))_2| = 6$.
Applying the zero padding would require the addition of $p = 00$ to the startning of both the sequences $(Q_0)_2$ and $(c(X))_2$ so that the resultant padded stored binary sequence would be

$$0010101000111111.$$

When reconstructing the unary $\oplus$-FA from this binary sequence we note that the first two states have no incoming transitions as $(c(X))_2 = 00111111$ and are not initial states as $(Q_0)_2 = 00101010$ so that they can be removed as initial-useless. Thus the unary $\oplus$-FA can be reconstructed as in example 6.3 (page 146).

This concludes our creation of the storage format. In the next section we discuss how a unary $\oplus$-FA stored in the above format can be executed in an application without reconstruction.

6.1.3 Application

We have constructed a storage format with a good compression ratio. However, we are not only interested in the compression ratio of the storage format, but also how the stored $\oplus$-FA can be used in an application. In the previous section we showed how the storage format could be used to reconstruct the stored unary $\oplus$-FA with one final state. In this section we show how a unary $\oplus$-FA in this format can be used to emulate the $\oplus$-FA without reconstructing it.

Given a unary $\oplus$-FA in the format of the binary sequence

$$\langle p\ ,\ (Q_0)_2\ ,\ p\ ,\ (c(X))_2\rangle\,,$$

we wish to

1. be able to calculate the series of active state sets, and
2. be able to calculate whether or not the current active state set contains the final state.

The initial active state set is the initial state set which is represented by the binary sequence $(Q_0)_2$ in the storage format. Each bit in $(Q_0)_2$ represents a state in the unary $\oplus$-FA. A state is active in the initial state set if and only if the corresponding bit is 1. The other element of the storage format is the sequence $(c(X))_2$ which is a binary representation of the transitions from the final state q_{n-1} to the other states.

To emulate the unary $\oplus$-FA with one final state we need only to store these two binary sequences and use three computer operations on binary sequences. These computer operations are

1. a right shift[5] (SH),
2. scalar multiplication, and
3. the symmetric difference of two sequences.

These operations will allow us to calculate the sequence of active state sets $S(0), S(1), \ldots$ where $S(i) = s(i)_0 s(i)_2 \ldots s(i)_{n-1}$. A state q_j is active in the active state set, $S(i)$, if and only if $s(i)_j = 1$. A state sequence after i alphabet inputs, $S(i)$, can be calculated from $S(i-1)$ as follows:

$$S(i) \ = \ \mathrm{SH}(S(i-1)) \oplus s(i-1)_{n-1}.(c(X))_2.$$

The right shift operation emulates the transitions from the non-final states, q_i, to the following state, q_{i+1}. The scalar multiplication operation emulates the transition from the final state to the states in the unary $\oplus$-FA. The symmetric difference operation emulates the interaction of the transitions.

As only the q_{n-1} is a final state, the binary value $s(i)_{n-1}$ for each active state set $S(i)$ indicates whether or not it contains a final state.

In the following example we examine how unary $\oplus$-FA can be used in an application.

[5] $\mathrm{SH}(x)$, where $x = x_0 x_1 \ldots x_k$ results in the sequence $0x_0 x_1 \ldots x_{k-1}$

Example 6.5 (Unary Application)
Using the unary $\oplus$-FA M from example 6.4 we have that M can be stored as the binary sequence

$$0010101000111111,$$

which we divide into the two sequences representing the active state set, $S_0 = 00101010$ and the characteristic polynomial $c(X) = 00111111$, from which we can run M as shown in the following table:

i	$S(i) = \mathrm{SH}(S(i-1)) \oplus s(i-1)_{(n-1)}.c(X)$	$\mathrm{SH}(S_i)$	$s(i)_{n-1}.c(X)$	output
0	00101010	00010101	00000000	0
1	00010101	00001010	00111111	1
2	00110101	00011010	00111111	1
3	00100101	00010010	00111111	1
4	00101101	00010110	00111111	1
5	00101001	00010100	00111111	1
6	00101011	00010101	00111111	1
7	00101010	00010101	00000000	0

.

The sequence $S(0), S(1), \ldots, S(6)$ corresponds to the state sequence

$$\{q_0, q_2, q_4\} \to \{q_1, q_3, q_5\} \to \{q_0, q_1, q_3, q_5\} \to \{q_0, q_3, q_5\} \to \{q_0, q_2, q_3, q_5\} \to \{q_0, q_2, q_5\} \to \{q_0, q_2, q_4, q_5\} \to \{q_0, q_2, q_4\} \to \cdots$$

This corresponds to the active state set sequence of the original unary $\oplus$-FA from example 5.17 (page 91).

The application of a unary $\oplus$-FA in this format requires the storage of the current active state set, characteristic function, and the zero padding , p. The current active state set and characteristic function each use n bits of storage. The zero padding p for each subsequences of the storage format can consist of most seven bits each as they increase the size of the subsequence to lengths of eight bits. Each round of operation requires at most one scalar multiplication, one shift and one addition operation. So that yhe algorithm requires at most $2n + 14$ bits storage space and 3 sequence operations per round to emulate the unary $\oplus$-FA with one final state.

This file format is not easily extendable to use with non-unary $\oplus$-FAs as we have multiple possible transition sequences depending on the input symbol. Instead of extending this algorithm, in the next section we discuss an alternative compressed format for the general $\oplus$-FA.

6.2 Non-unary $\oplus$-FA

In this section we wish to develop a storage format for general $\oplus$-FA. We base the storage format for general $\oplus$-FA on a storage format for DFAs from [4] with some compression techniques based on the properties of $\oplus$-FAs.

We start by discussing the basic storage format for general $\oplus$-FAs and show how to execute this basic format. After that, we discuss various compression methods for the basic format. Throughout this chapter we provide the following running example to highlight the concepts we are discussing.

Example 6.6

Let $M = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{0, 1\}, \delta, \{q_0\}, \{q_4, q_5\}, \oplus)$ *be a* $\oplus$*-FA with* δ *given by (also shown in figure 6.2)*

δ	0	1
q_0	$\{q_0, q_1\}$	$\{q_2\}$
q_1	$\{q_2\}$	$\{q_1, q_3, q_5\}$
q_2	$\{q_3\}$	$\{q_0\}$
q_3	$\{q_3, q_4\}$	$\{q_0, q_2\}$
q_4	$\{q_3\}$	$\{q_0\}$
q_5	$\emptyset$	$\emptyset$

.

Before we start discussing the basic storage format, it is important to note the following: we developed the unary $\oplus$-FA storage format as a binary sequence of arbitrary length and zero padded it to be usable by a computer. The general $\oplus$-FA storage format we develop in this section stores the data in fixed lengths. Although this does not result in the most compact representation of the $\oplus$-FA, it does allow for efficient addressing of states and transitions as we shall discuss later.

6.2.1 Basic Format

The basic format is taken from the storage format for DFAs from [4]. The basic is to describe each state $q \in Q$ in a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ as a series of its outgoing transitions. An outgoing transition from a given state can be described as the alphabet symbol and destination state.

So, for the $\oplus$-FA M from example 6.6 (page 152), the state q_3 would be represented by the tuples set, $< \sigma, \delta >$,

$$\{ < 0, q_3 >, < 0, q_4 >, < 0, q_0 >, < 0, q_2 > \}.$$

We can store the entire transition function by representing each state as its transitions and listing their sequences in a predetermined order.. We distinguish the transition sequence from a state from a transition sequence from another state by marking the starting and end of the transition sequences in the transition function.

Example 6.6's transition function can be stored as follows (where the

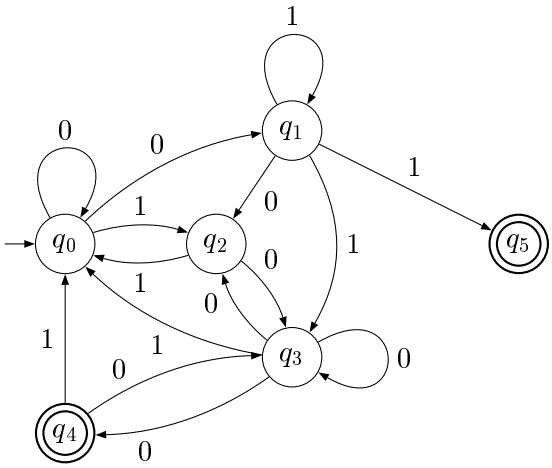

Figure 6.2: Example 6.6.

lines represent the separation of the transition sequences):

σ	δ
0	q_0
0	q_1
1	q_2
0	q_2
1	q_1
1	q_3
1	q_5
0	q_3
1	q_0
0	q_3
0	q_4
1	q_0
1	q_2
0	q_3
1	q_0
0	$\emptyset$

.

Given the order that the state transition sequences are listed in, we can determine a specific transitions from a state by making use of the state markers. Therefore, if we wish to find the transitions of the state q_3 we simply search the third marker in the stored data sequence. Similarly, to find the transitions of a destination state we can make use of the number of markers between the transition sequences of the current and destination state. Therefore, a transition from the state q_1 to q_3 would require a jump of two markers to find the transitions of the destination state, q_3. However, the detection of the markers requires computation to find the exact position of the various markers. Instead of storing the destination state in this way we can replace the label of the destination state with the address of the marker. This enables an application to jump directly to the transition sequence of the destination state instead of manually searching for it.

As memory is accessed in fixed bytes by a computer, it is faster to jump to an address rather than searching for a specific bit. To make addressing

easier we define each transition, including the alphabet symbol, to be of fixed length in bytes. In this way we only reference memory in bytes and for searching purposes need not worry about bits. We use four bytes in total for each transition with three bytes reserved for the address. The remaining byte is reserved for the alphabet symbol and flags which will be defined at a later stage. The bytes are labeled from address[6] 000001_{16} which corresponds to the state q_0.

An addressing problem arises when trying to store a state with no transitions in this way. An initial solution to this problem would be to remove transitions to this state and remove it from the $\oplus$-FA. However, if the state is final we need to not only store such a state, but ensure that it is distinguishable from other final states with no transitions. In such a case the final state can be stored with a dummy transition to the address 000000_{16}. An application would recognize and ignore these transitions when encountered.

Applying these changes to the $\oplus$-FA M from example 6.6, the resultant stored sequence would be:

Address	σ	δ (Pointers)
000001_{16}	0	000001_{16}
000005_{16}	0	$00000D_{16}$
000009_{16}	1	$00001D_{16}$
$00000D_{16}$	0	$00001D_{16}$
000011_{16}	1	$00000D_{16}$
000015_{16}	1	000025_{16}
000019_{16}	1	$00003E_{16}$
$00001D_{16}$	0	000025_{16}
000021_{16}	1	000001_{16}
000025_{16}	0	000025_{16}
000029_{16}	0	000035_{16}
$00002D_{16}$	1	000001_{16}
000031_{16}	1	$00001D_{16}$
000035_{16}	0	000025_{16}
000039_{16}	1	000001_{16}
$00003E_{16}$	0	000000_{16}

.

From the above structure we can determine the state set, alphabet and transition function of the stored $\oplus$-FA. This means that only the initial and final state sets are still to be stored.

We handle the initial state set by using the initial state transformation from theorem 5.42 (page 130). Applying this transformation to the $\oplus$-FA to be stored we obtain an equivalent $\oplus$-FA with the same number of states and exactly one initial state[7]. Transforming a $\oplus$-FA in this way before applying the storage format we can implicitly store the single initial state as the first state in the transition sequences.

[6]Note that we use the hexadecimal system for the addresses, denoted as x_{16}, where x is a hexadecimal number.

[7]Note that any change to the $\oplus$-FAs structure would require the stored data to be changed.

To handle the final state set we borrow an idea from Output FAs[8]. In Output FAs, an output is stored on the transitions rather than labeling the states as final. By labeling the outputs with either a 0 representing a non-final destination state or a 1 representing a final destination state we can store the finality property of a state in the transitions to the state rather than in the state itself. This means that in the storage format we need not store whether a state is final explicitly with the state, but can instead implicitly store this property of the state on all of the transitions to the state. This may seem counter intuitive as we are adding extra data to the stored format, but as shown later it can be used to help compress the amount of stored data.

A problem arises when the initial state is a final state. This case is solved by using a preamble byte in the storage format. The final bit of this preamble byte represents the finality property of the initial state. This bit is 1 if the initial state is final and 0 otherwise.

By applying the initial state transformation and making use of a final state marker, we can store all five properties of a $\oplus$-FA. Before providing a formal definition of the basic format we need to formalize the final destination state marker and the markers representing the separation of states. We do this by reserving the first two bits of the transition as markers for the separation of states and the finality property of the destination state respectively. We define these reserved bits as the last and final flags.

Storing the markers in this way reduces the number of bits representing the alphabet symbol to 6 bits. However, most applications for $\oplus$-FA will not need an alphabet with more than 64 symbols which can map into the remaining six bits[9].

We define the basic format for $\oplus$-FA as follows:

Format 6.7 (Basic Format)

Given a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$ with $Q = \{q_0, q_1, \ldots, q_{n-1}\}$, $\Sigma = \{\sigma_0, \ldots, \sigma_{m-1}\}$ and $Q_0 = \{q_0\}$ we can store M as follows:

$$\langle \textbf{PREAMBLE}, \textbf{TRANSITIONS} \rangle$$

*where we define the **PREAMBLE** as:*

$$(0000000x)_2$$

with $x = 1$ if $q_0 \in F$ and 0 otherwise.

We define the **TRANSITIONS** *as the ordered sequence of transitions $\delta(q_i, \sigma) = q_j$ for $i = 0 \ldots n-1$ and $j = 0 \ldots m-1$ where each transition is*

[8] See definition A.9, page 170.

[9] We can increase the alphabet size by increasing the number of bytes used for the alphabet, but for use in this thesis we formalize the alphabet to using at most 6 bits. That equates to $2^6 = 64$ alphabet symbols.

Address	*Last Flag*	*Final Flag*	σ	δ	*Hexadecimal*	*Description*
000000_{16}		0			00	*Preamble*
000001_{16}	0	0	0	000001_{16}	00000001_{16}	$\delta(q_0, 0) = q_0$
000005_{16}	0	0	0	$00000D_{16}$	$0000000D_{16}$	$\delta(q_0, 0) = q_1$
000009_{16}	1	0	1	$00001D_{16}$	$8100001D_{16}$	$\delta(q_0, 1) = q_2$
$00000D_{16}$	0	0	0	$00001D_{16}$	$0000001D_{16}$	$\delta(q_1, 0) = q_2$
000011_{16}	0	0	1	$00000D_{16}$	$0100000D_{16}$	$\delta(q_1, 1) = q_1$
000015_{16}	0	0	1	000025_{16}	01000025_{16}	$\delta(q_1, 1) = q_3$
000019_{16}	1	1	1	$00003E_{16}$	$C100003E_{16}$	$\delta(q_1, 1) = q_5$
$00001D_{16}$	0	0	0	000025_{16}	00000025_{16}	$\delta(q_2, 0) = q_3$
000021_{16}	1	0	1	000001_{16}	81000001_{16}	$\delta(q_2, 1) = q_0$
000025_{16}	0	0	0	000025_{16}	00000025_{16}	$\delta(q_3, 0) = q_3$
000029_{16}	0	1	0	000035_{16}	40000035_{16}	$\delta(q_3, 0) = q_4$
$00002D_{16}$	0	0	1	000001_{16}	01000001_{16}	$\delta(q_3, 1) = q_0$
000031_{16}	1	0	1	$00001D_{16}$	$8100001D_{16}$	$\delta(q_3, 1) = q_2$
000035_{16}	0	0	0	000025_{16}	00000025_{16}	$\delta(q_4, 0) = q_3$
000039_{16}	1	0	1	000001_{16}	81000001_{16}	$\delta(q_4, 1) = q_0$
$00003E_{16}$	1	0	0	000000_{16}	80000000_{16}	$\delta(q_5, 0) = \emptyset$

Figure 6.3: Example 6.8.

stored in the format of a **TRANSITION** *as follows:*

$$(Last\ Flag,\ Final\ Flag,\ Alphabet\ Symbol(6\ bits),\ Address(3\ bytes))_2$$

Example 6.8
Let M be the $\oplus$-FA from example 6.6. The basic format representation of M is given in figure 6.3

Note 16 *For comparison later we note that the $\oplus$-FA from the running example is stored using 65 bytes in this basic format.*

Before discussing compression of the basic format we show how the basic format can be used to execute the $\oplus$-FA.

Where the implementation of a unary $\oplus$-FA stored a binary sequence representing the active state set and a binary sequence representing the transitions, the implementation of the basic format for $\oplus$-FAs stores the $\oplus$-FA in the basic format and the addresses of the active states. On input of a symbol, we will generate the new active state set by comparing the transitions of the active states. Similarly, we will generate the subset of active final states which we shall call the final active state set. The transitions from every address stored in the previous active state set is processed until a transition with the last flag set is found. Every transition processed with the corresponding input symbol is handled as follows:

- If the new active state set does not contain the destination address then the address is added to the new active state set, otherwise the address is removed from the new active state set.
- Similarly, if the transition has the final flag set to one the destination state is added to ore removed from the final active state set depending on wether the state is already an element.

If, after all the transitions have been processed, the final active state set is not empty, the active state set in the original $\oplus$-FA contains a final state after the given input sequence.

Using the above description, the running example can be implemented as shown in the following example:

Example 6.9

Let M be the $\oplus$-FA stored in the basic format as shown in example 6.8 (page 156). On input of the word 00101 *M would be executed as follows:*

σ	*Active State Addresses*	*Active State Set*	*Final*
	$\{000001_{16}\}$	$\{q_0\}$	No
0	$\{000001_{16}; 00000D_{16}\}$	$\{q_0, q_1\}$	No
0	$\{000001_{16}; 00000D_{16}; 000001D_{16}\}$	$\{q_0, q_1, q_2\}$	No
1	$\{000001_{16}; 00000D_{16}; 000001D_{16}; 000025_{16}; 00003E_{16}\}$	$\{q_0, q_1, q_2, q_3, q_5\}$	Yes
0	$\{000001_{16}; 00000D_{16}; 000001D_{16}; 000035_{16}\}$	$\{q_0, q_1, q_2, q_4\}$	Yes
1	$\{00001D_{16}; 00000D_{16}; 0000025_{16}; 00003E_{16}\}$	$\{q_1, q_2, q_3, q_5\}$	Yes

.

As the active state set can consist of at most all the states in the $\oplus$-FA, the amount of memory needed during execution of the basic format has a complexity of $\mathcal{O}(n)$ with $n = |Q|$ above the size of the stored $\oplus$-FA. The running complexity of the above algorithm is based on the number of transitions processed as well as the operations involved in the addition and removal of the active states. An implementation of this format would require at most $\log_2(n)$ operations to find the destination state in the new active state set and at most m iterations of this search where m is the number of transitions in the $\oplus$-FA. This results in a running complexity of $\mathcal{O}(m\log_2(n))$.

6.2.2 Compression algorithms

Given the basic format as presented in section 5.1.3, we now discuss three compression algorithms based on the DFA storage format used for [4]. These improvements on the basic format exploit shared transitions, relative addresses and variable address length respectively.

Shared Transitions

As discussed in section 5.2.1 (page 107), states in a $\oplus$-FA may be equivalent or contained and not be reducible. Thus, although a $\oplus$-FA may be minimal, it can consist of states whose transitions are completely contained within the transitions from another state. One such case is where two states, of which at least one is final, share all their transitions. The shared transitions compression algorithm attempts to reduce the size of the stored $\oplus$-FA by storing series of transitions together.

This can be done by removing the state whose transitions form a subset of another states transitions. The removed state will be referenced within the transitions of the remaining state. We firt list the transitions which are not shared, and then the shared transitions. By referencing the address of the first of the transitions we can process all of the larger transitions from a state. Alternatively, by referencing the first of the shared transitions, we can process all of the contained transitions from a state.

Therefore, in a $\oplus$-FA $M = (Q, \Sigma, \delta, Q_0, F, \oplus)$, if for every $\sigma \in \Sigma$ it holds that $\delta(q_i, \sigma) \subseteq \delta(q_j, \sigma)$ with $q_i, q_j \in Q$, the state q_i can be stored inside the state q_j by first storing all the non shared transitions $\delta(q_j, \sigma) \oplus \delta(q_i, \sigma)$ and then the transitions $\delta(q_i, \sigma)$ for every $\sigma \in \Sigma$.

In example 6.8 (page 156) we have that the transitions of the states q_4 and q_3 are

δ	0	1
$3q_3$	$\{q_3, q_4\}$	$\{q_0, q_2\}$
q_4	$\{q_3\}$	$\{q_0\}$

.

From this we can verify that the transitions of the state q_4 are contained within the transitions of the state q_3.

We can store the states transitions in the following order:

σ	δ
0	q_4
1	q_2
0	q_3
1	q_0

.

As mentioned above, two states may share all of their transitions, but not be reducible. This occurs if at least one of the states is final. Although we can store these states together we need to be able to distinguish the final states from each other. For this reason we cannot store final states which share all of their transitions together as a single address. However, we may store non-final states, which share transitions with a final states, together with the final state. In the running example we have that the states q_2 and q_4 share all of their transitions. As the state q_2 is non-final its transitions may be stored with the transitions of the state q_4.

Applying the shared transitions compression to the running example we have:

Example 6.10

Let M be the $\oplus$-FA from example 6.6 (page 152). Using the base format from example 6.8 (page 156) we combine the stored transitions of the states q_2 and q_4 inside the transitions of the state q_3, which results in the following stored data:

Address	*Last Flag*	*Final Flag*	σ	δ	*Hexadecimal*	*Description*
000000_{16}		0			00	*Preamble*
000001_{16}	0	0	0	000001_{16}	00000001_{16}	$\delta(q_0, 0) = q_0$
000005_{16}	0	0	0	$00000D_{16}$	$0000000D_{16}$	$\delta(q_0, 0) = q_1$
000009_{16}	1	0	1	000025_{16}	81000025_{16}	$\delta(q_0, 1) = q_2$
$00000D_{16}$	0	0	0	000025_{16}	00000025_{16}	$\delta(q_1, 0) = q_2$
000011_{16}	0	0	1	$00000D_{16}$	$0100000D_{16}$	$\delta(q_1, 1) = q_1$
000015_{16}	0	0	1	$00001D_{16}$	$0100001D_{16}$	$\delta(q_1, 1) = q_3$
000019_{16}	1	1	1	$00002D_{16}$	$C100002D_{16}$	$\delta(q_1, 1) = q_5$
$00001D_{16}$	0	1	0	000025_{16}	40000025_{16}	$\delta(q_3, 0) = q_4$
000021_{16}	1	0	1	000025_{16}	81000025_{16}	$\delta(q_3, 1) = q_2$
000025_{16}	0	0	0	$00001D_{16}$	$0000001D_{16}$	$\delta(q_3, 0) = q_3$
000029_{16}	1	0	1	000001_{16}	81000001_{16}	$\delta(q_3, 1) = q_0$
$00002D_{16}$	1	0	0	000000_{16}	80000000_{16}	$\delta(q_5, 0) = \emptyset$

.

The $\oplus$-FA from the running example using the shared transition compression is stored using 49 bytes, resulting in a size difference of 16 bytes or 24.6%.

The structure changes caused by applying the shared transitions compression to the base format does not impact on the execution of the $\oplus$-FA. However, it may occur that an address will be an element of the final active state set, but not of the active state set, due to final states being stored with non-final states.

Using the input sequence from example 6.9 (page 157) the stored data in example 6.10 will run as follows:

Example 6.11

Let M be the $\oplus$-FA stored in the format as shown in example 6.10 (page 159). On input of the word 00101 *M would be executed as follows:*

σ	*Active State Addresses*	*Active State Set*	*Final*
	$\{000001_{16}\}$	$\{q_0\}$	*No*
0	$\{000001_{16}; 00000D_{16}\}$	$\{q_0, q_1\}$	*No*
0	$\{000001_{16}; 00000D_{16}; 0000025_{16}\}$	$\{q_0, q_1, q_2\}$	*No*
1	$\{000001_{16}; 00000D_{16}; 000001D_{16}; 000025_{16}; 00002D_{16}\}$	$\{q_0, q_1, q_2, q_3, q_5\}$	*Yes*
0	$\{000001_{16}; 00000D_{16}\}$	$\{q_0, q_1, q_2, q_4\}$	*Yes*
1	$\{00000D_{16}; 00001D_{16}; 0000025_{16}; 00002D_{16}\}$	$\{q_1, q_2, q_3, q_5\}$	*Yes*

.

Relative Addressing

The second compression algorithm is that based on relative addresses. In the storage format the sequence in which the states of the $\oplus$-FA are ordered can be done arbitrarily. However, we can and usually do order these states in such a way that states which have transitions which connect them are grouped together. Relative addressing compression takes advantage of this fact. We currently reference the destination state of a transition by its absolute address; that is, its offset from the beginning of the file. This absolute address currently takes up three bytes allowing for 2^{24} distinct byte addresses or 2^{22} distinct states[10]. Relative addressing attempts to compress the $\oplus$-FA by allowing transitions to reference the destination state by the amount of bytes between the address of the transition and the address of the destination state. We define this relative addressing to account for gaps of at most 2^6 bytes, by using six bits to determine the magnitude and a seventh bit to assign the sign (positive or negative). The remaining eighth bit is used as a marker representing whether or not the address is relative or absolute.

For example in example 6.10 (page 159) we have a transition from the state q_3 to the state q_0. This transition is stored at the address $00001D_{16}$ and the state q_0 is stored at the address 000001_{16}. The difference between the addresses of the state q_0 and the transition would be $000001_{16} - 00001D_{16} = -00001C_{16}$. Therefore, in the example the transition could be stored using a relative address pointer of $-1C_{16}$

We can distinguish relative addresses from absolute addresses by a assigning a value to a specified bit. This is similar to the the flags representing final states and the last transition of a state. We assign the first bit of the second byte of a transition to mark whether the pointer is absolute, 0, or relative, 1.

If the pointer is relative only two bytes are used for the transition with the final seven bits representing the distance and direction of the destination state. If the pointer is absolute, four bytes are used for the transition with the address space reduced to 2^{23} bytes. However, as some transitions may be stored as two bytes instead of four bytes, the number of transitions this format may contain remains at 2^{22} transitions.

Although the possible number of transitions remains constant, the file size will decrease by two bytes for every relative addressed transition. Before showing an example of relative addressing we present two notes.

Note 17 *It is important to note that any transition changed from an absolute address to relative address modifies the addresses of the transitions stored after this transition, thus affecting all transitions to addresses after*

[10] This is due to the fact that currently each state is stored using four bytes.

the relative addressed transition. This may in turn allow for more destination states to be within a distance of 2^6 *bytes allowing for relative addressing providing further compression.*

Note 18 *It is important to note that the dummy transitions to the state* 000000_{16} *remain as absolute addresses so that they may be easily picked up by the application.*

Applying relative addressing to the running example we have:

Example 6.12

Let M be the ⊕-FA from example 6.8 (page 156). We use the base format with the shared transition compression applied from example 6.10 (page 159). By applying the relative addressing we have that as the greatest address referenced in the ⊕-FA M is smaller than 2^6*, all the transitions can be made relative resulting in the following:*

Address	*Last Flag*	*Final Flag*	σ	*Rel*	δ	*Hexadecimal*	*Description*
000000_{16}		0				00	*Preamble*
000001_{16}	0	0	0	1	$+00_{16}$	0080_{16}	$\delta(q_0, 0) = q_0$
000003_{16}	0	0	0	1	$+04_{16}$	0084_{16}	$\delta(q_0, 0) = q_1$
000005_{16}	1	0	1	1	$+0F_{16}$	$818F_{16}$	$\delta(q_0, 1) = q_2$
000007_{16}	0	0	0	1	$+0D_{16}$	$008D_{16}$	$\delta(q_1, 0) = q_2$
000009_{16}	0	0	1	1	-02_{16}	$01C2_{16}$	$\delta(q_1, 1) = q_1$
$00000B_{16}$	0	0	1	1	$+04_{16}$	0184_{16}	$\delta(q_1, 1) = q_3$
$00000D_{16}$	1	1	1	1	$+0A_{16}$	$C18A_{16}$	$\delta(q_1, 1) = q_5$
$00000F_{16}$	0	1	0	1	$+04_{16}$	4084_{16}	$\delta(q_3, 0) = q_4$
000011_{16}	1	0	1	1	$+02_{16}$	8182_{16}	$\delta(q_3, 1) = q_2$
000013_{16}	0	0	0	1	-04_{16}	$00C4_{16}$	$\delta(q_3, 0) = q_3$
000015_{16}	1	0	1	1	-14_{16}	$81D4_{16}$	$\delta(q_3, 1) = q_0$
000017_{16}	1	0	0	0	000000_{16}	80000000_{16}	$\delta(q_5, 0) = \emptyset$

where rel determines whether the transition is relative(1) or absolute(0).

The ⊕-FA from the running example using relative addresses is stored using 27 bytes, a size difference of 22 bytes from the shared transitions compression or a 45%. This is an effective 58.5% from the the basic format (65 bytes).

The structure changes caused by applying the relative addressing compression requires the execution of the stored ⊕-FA to be modified. That is, after reading the alphabet symbol, last and final flags of the transition. The application must read the first bit of the second byte to determine whether or not the address is absolute or relative. If the address is absolute it is handled normally. Otherwise, the second bit is read to determine if the destination address occurs before or after the current address, after which the remaining bits are read to determine how far back or forward the destination address is.

Using the input sequence from example 6.9 (page 157) the stored data in example 6.12 will run as follows:

Example 6.13

Let M be the $\oplus$-FA stored in the format as shown in example 6.12 (page 161). On input of the word 00101 *M would be executed as follows:*

σ	*Active State Addresses*	*Active State Set*	*Final*
	$\{000001_{16}\}$	$\{q_0\}$	*No*
0	$\{000001_{16}; 000007_{16}\}$	$\{q_0, q_1\}$	*No*
0	$\{000001_{16}; 000007_{16}; 0000025_{16}\}$	$\{q_0, q_1, q_2\}$	*No*
1	$\{000001_{16}; 000007_{16}; 000000F_{16}; 000013_{16}; 000017_{16}\}$	$\{q_0, q_1, q_2, q_3, q_5\}$	*Yes*
0	$\{000001_{16}; 000007_{16}\}$	$\{q_0, q_1, q_2, q_4\}$	*Yes*
1	$\{000007_{16}; 00000F_{16}; 0000013_{16}; 000017_{16}\}$	$\{q_1, q_2, q_3, q_5\}$	*Yes*

.

Variable Address Length

The final compression algorithm generalizes the size of the absolute addresses. That is, it allows the stored $\oplus$-FA to determine how many bytes will be needed to store the addresses of the transitions in the $\oplus$-FA. The worst case scenario would be where every state was stored as an absolute transition. In this case, if the $\oplus$-FA had m transitions, each transition would require a unique address so that with b equal to the number of bits used to store a transition (including the alphabet symbol) we have that

$$2^{8(b-1)-1} \geq b.m + 1,$$

so that

$$\frac{2^{8(b-1)-1} - 1}{b} \geq m.$$

This can be calculated for m by using a brute force search of b until a satisfactory b is found. By storing this value b as the remaining 7 bits of the preamble, so that $b \leq 2^7$, the format can allow for up to 2.7×10^{303} transitions. By increasing the preamble to more bytes the number of transitions stored can increase indefinitely, up to the memory capacity of the machine involved.

The modification of the size of the absolute addresses does not change the size of the relative addresses, although this could be modified with respect to the absolute address. The number of bytes saved by the relative address will decrease with the increase in size of the relative address space.

Other compression algorithms

We have provided three compression algorithms for the basic format. These were the compression algorithm we investigated which could be adapted for use with $\oplus$-FAs. There exist other compression algorithms which work on DFAs and NFAs but do not function correctly on $\oplus$-FA due to the interaction of transitions.

We finish this chapter by discussing the experimental results of the algorithm in the next section.

6.2.3 Experimental Analysis

We tested the storage format with all the compression algorithms for $\oplus$-FAs on randomly generated $\oplus$-FAs. The results discussed here are from a set of 1000 uncompressed $\oplus$-FAs with the alphabet $\Sigma = \{0, 1\}$ and 1000 states. The states were predefined and for each state both the number of states and destination states of these transitions were randomly generated.

The $\oplus$-FAs used absolute address lengths of 2 bytes, resulting in transitions of length 3 bytes.

On average the $\oplus$-FAs had:

- 7234 transitions, of which 433 were stored as relative;
- 2.3 states were contained by other states[11];
- 5.6 transitions per state; and
- an average file size of 15276 bytes.

Using only the basic format without compression the $\oplus$-FAs would require on average 28936 bytes per file resulting in a compression of 0.52%.

These results were influences by the following factors:

- No large $\oplus$-FAs used in practical circumstances were available;
- the small number of relative transitions: a higher number can be expected through sorting of states;
- the random generation of the $\oplus$-FAs was not based on any random FA generating models. Instead, number of transitions and destination states were generated randomly;

and as such cannot be taken as a full indication of the power of the storage format.

This finishes our discussion on storage formats for $\oplus$-FAs. We provided a format for both unary $\oplus$-FAs and general $\oplus$-FAs. The unary $\oplus$-FA storage format was based on the minimization algorithm from the previous section. This storage format had both an excellent compression ratio and execution computational complexity.

The general $\oplus$-FA storage format did not provide a constant compression ratio such as the unary $\oplus$-FA storage format. The compression ratio depended on the structure of the $\oplus$-FA and the number of shared transitions. In the case of $\oplus$-FAs, it is likely due to the nature of the interaction of transitions. The final format provided executed some compression algorithms. However, further compression algorithms can be found and the format provided here was made to be as general as possible to provide a good basis for

[11] Remember that a contained state shares all of it's transitions with another state.

further specific compressed storage formats. Note this storage format could be used for any $*$-FA.

It is of interest to note that both the unary $\oplus$-FA and the general $\oplus$-FA storage formats made use of transformations on the $\oplus$-FA to simplify the storage format. In the case of the unary storage format this was the minimization algorithm from the previous chapter. In the case of the general storage format it reduced the size of the initial state set to a single state.

Chapter 7

Conclusion

The goal of this thesis was to investigate state minimization and reduction algorithms for $\oplus$-FAs. The algorithms we presented were based on properties of $\oplus$-FAs, such as the state sequence and relationship that unary $\oplus$-FAs have with LFSRs and the subset $\oplus$ transformation, and minimization techniques for DFAs and NFAs, such as Kameda and Weiner's NFA minimization technique.

Chapter 2 provided basic definitions and properties such as equivalence of FAs. This chapter not only provided the basic concepts, but also the notations and format that would be used throughout this thesis. In chapter 3 we defined $*$-FAs as well as the properties of such FAs. We rounded out these definitions with examples of both $\cap$-FAs and $\oplus$-FAs. We extended the properties of NFAs and DFAs in chapter 4 to include the concepts of predecessors and successors. Using these concepts we defined and determined reducible states, which could be removed from the FAs to reduce the number of states. Using these reducible states we discussed three DFA minimization algorithms based upon the three categories of minimization algorithms namely: state comparison; language reconstruction and transformation reconstruction. The minimization of DFAs was shown to be simpler than that of NFA minimization due to DFAs not containing reducible states, as well as having at most one transition per state per alphabet symbol. The three minimization algorithms we discussed was that of dictionary minimization [4](section 4.2.1), a language reconstruction algorithm based on a specialized form of DFAs, the TRIE. The second minimization algorithm we discussed was that of Hopcroft [7](section 4.2.2), a state comparison algorithm. This algorithm proved to be the fastest running with a worst case computational complexity of $\mathcal{O}(n\log(n))$. The final DFA minimization algorithm we discussed was that of Brzozowski [2](section 4.2.3), a transformation construction algorithm. Although this algorithm runs with a computational complexity of $\mathcal{O}(2^n)$ it is of interest due to it's use of the subset constriction technique to reduce reducible states without detecting them.

These three algorithms were used as a base to discuss a NFA minimization and a NFA reduction algorithm. We began the discussion of NFA reduction by discussing reducible states belonging to NFAs but not DFAs. The NFA minimization algorithm we discussed was that of Kameda and Weiner [11](section 4.3.1), a transformation construction technique which used a combination of Brzozowski's DFA minimization algorithm the subset construction algorithm to generate a method to construct a minimal NFA from the minimal DFA. Although guaranteed to give a minimal NFA, the intersection rule required a subset assignment to construct the minimal NFA. By making use of state maps Kameda and Weiner presented a method of obtaining a valid subset assignment based on a cover map. However, this algorithm is still NP hard as it makes use of subset construction. As shown in [10] all NFA minimization algorithms are NP hard, and as such will always be impractical to perform.

For this reason Ilie an Yu[9](section 4.3.2) present an algorithm which does not minimize an NFA but rather reduce it. By simplifying the result to be reduced and not minimal, Ilie and Yu presented an algorithm which runs in polynomial time. This algorithm was proposed not only as a way to provide a smaller NFAs but also as a precursor to subset construction and NFA minimization techniques to improve the computational complexity.

Using these techniques from chapter 4 we approached the minimization and reduction of $\oplus$-FAs in chapter 5. We began this chapter by examining unary $\oplus$-FAs(section 5.1). Using the relationship between unary $\oplus$-FAs and LFSRs [5] we constructed a minimization algorithm which first calculated a unary $\oplus$-FAs final state sequence, using this sequence we applied a modification of the Berlekamp-Massey to generate the characteristic polynomial of minimal degree which generated the final state sequence. Using both this characteristic polynomial and final state sequence we were able to construct the minimal unary $\oplus$-FA with exactly one final state, but multiple initial states. After the initial discussion we provided examples of different types of the final state sequences and an equivalent minimal unary $\oplus$-FAs with on final state. We also showed that although this algorithm was adaptable to include multiple final states, the increase in computational complexity was too high.

In section 5.2 we discussed the minimization and reduction of non-unary $\oplus$-FAs. We began by discussing the concepts of reducible state in $\oplus$-FAs. We found that although initial-useless, final-useless and redundant state were still reducible, this was not true for all equivalent or contained states. This proved to be a problem when attempting to construct minimization and reduction techniques. Furthermore, due to the definitions of the regular expression and the characteristics of the $\oplus$ operator, language reconstruction algorithms would be inefficient to construct. The first minimization algorithm we provided for non-unary $\oplus$-FAs was that of subset reconstruction(section 5.2.3) based on Kameda and Weiner's intersection rule. We

generalized the subset reconstruction for all $*$-FAs and provided methods and examples to reconstruct $\cap$-FAs and $\oplus$-FAs from the minimal DFA. However, attempting to construct a suitable subset assignment involved use of the dual, which we could not determine efficiently. Thus although subset reconstruction guaranteed a minimal $\oplus$-FA a brute force search would be needed to find a valid subset assignment.

Lastly we examined possible state comparison algorithms for the minimization and reduction of $\oplus$-FAs. Due to the act that not all equivalent states are reducible in a $\oplus$-FA we could not construct a reduction algorithm similar to Ilie and Yu's NFA reduction algorithm. However, we were able to construct a transformation known as the subset $\oplus$ transformation. This transformation produced equivalent $\oplus$-FAs with the same number of states. Using this transformation we showed that for every $\oplus$-FA an equivalent $\oplus$-FA with exactly one initial state and the same number of states existed. We also used this transformation to transform reducible equivalent states into final-useless states, and reducible states into initial-useless states. The subset $\oplus$ transformation could effectively reduce the number of states in a $\oplus$-FA. Similarly, the subset $\oplus$ transformation could be used to reduce the number of transitions in a $\oplus$-FA.

In the final chapter, chapter 6 we discussed compression of $\oplus$-FAs. This had to deal with the storage of $\oplus$-FA in such a way that they could be run efficiently and stored in a compressed format. In section 6.1 we presented a form at for unary $\oplus$-FA based on the unary $\oplus$-FA minimization algorithm. This resulted in a file consisting of $\lceil \frac{n}{4} \rceil$ bytes which could be run efficiently.

In section 6.2 we provided a basic storage format for $\oplus$-FAs which stored the states implicitly and the finality property of $\oplus$-FAs in the transitions. By making use of the initial state $\oplus$ transformation, the initial states could be stored as the first state reducing the size of the file. We then presented three compression algorithms namely shared transitions, relative addressing and variable address lengths. These compression algorithms could theoretically drastically reduce the size of the stored $\oplus$-FA.

The work we have presented here is far from a conclusive study into the minimization of $\oplus$-FAs. We finish off this thesis by discussing possible future work into the minimization of $\oplus$-FAs.

Future Work

Our research of $\oplus$-FA minimization algorithms is hampered by the scarcity of research into $\oplus$-FAs. However, we believe that the algorithms, as discussed here, provide a good basis for further research, not only into $\oplus$-FA but also into $*$-FA minimization.

The unary $\oplus$-FA algorithm we discussed in chapter 5 generated a minimal unary $\oplus$-FA with one final state from a unary $\oplus$-FA. We discussed possibly extending this algorithm to calculate a minimal $\oplus$-FA with a non-

unary alphabet and/or more than one final state. The method which we investigated to extend this algorithm to multiple final states proved to be inefficient. However, other methods may exist in which to extend this algorithm to $\oplus$-FAs with a non-unary alphabet and/or more than one final state. Such methods may include different forms of mapping $\oplus$-FA to other sorts of logical machines and making use of their appropriate minimization algorithms.

In the case of general $\oplus$-FA minimization algorithms, we presented one minimization algorithm based on reconstruction from an equivalent DFA. Although this algorithm works, it failed to present a method to calculate a subset assignment which would generate the minimal $\oplus$-FA. Kameda and Weiner's NFA minimization algorithm made use of the dual to calculate these subset assignments for the case of traditional NFAs. Research into finding an efficient method which could generate an appropriate subset assignment for $\oplus$-FA reconstruction would be of great benefit to this algorithm. Furthermore, research into generating the dual of a $\oplus$-FA may lead to further reduction properties of $\oplus$-FAs.

As with NFA minimization algorithms, $\oplus$-FA minimization algorithms will always be NP-hard. Reduction algorithms can address this by ensuring faster computation through simpler comparisons but does not always result in the minimal FA. Our research into $\oplus$-FA reduction algorithms yielded the subset $\oplus$ transformation. This transformation provided a basis for a reduction algorithm as well as other beneficial results such as the initial state set transformation. The reduction algorithm we discussed was open ended in terms of implementation. As such, similar to [9] methods which make use of the subset $\oplus$ transformation could be made to be efficient for the reduction of $\oplus$-FAs. Further research into the subset $\oplus$ transformation itself may yield more properties of $\oplus$-FAs which could be used to simplify the reduction and minimization problem.

We finished this chapter with a brief discussion of subset reconstruction of $\cap$-FAs. It follows that the subset reconstruction algorithm could be a common point for $*$-FA minimization algorithms. Also any minimization algorithm which work for $\oplus$-FAs may be adapted for use with other $*$-FAs. However, as this thesis focused on the minimization of $\oplus$-FAs there is still a large area available fore research in the minimization of $*$-FAs.

Lastly we presented compressed file formats for both unary and general $\oplus$-FAs. The formats we presented were basic and meant as a basis for future work on $\oplus$-FA storage and compression.

We believe that the research we have presented provides both algorithms and a good basis for any further research into the minimization and reduction of both $\oplus$-FAs and $*$-FAs in general. Methods such as the subset $\oplus$ transformation and unary $\oplus$-FA minimization algorithms hold much promise for future research.

Appendix A

Terminology

Convention A.1 (Sequence)
*A **sequence** x is defined as a series of consecutive characters, that is*

$$x = x_0 x_1 x_2 \dots x_n,$$

where x_i is a single character.

Convention A.2 (Sets) *We use capital letters to denote **sets** and lowercase letters to denote **elements** of sets, as follows:*

$$Q = \{q_0, q_1, \dots, q_{n-1}\}.$$

*The **size** of a set Q with n elements is denoted as*

$$|Q| = n.$$

Definition A.3 (Set Operators)
Given two sets A and B, we use the following set operators:

- *the **union** set operator:*

$$A \cup B = \{x | x \in A \text{ or } x \in B\},$$

- *the **intersection** set operator:*

$$A \cap B = \{x | x \in A \text{ and } x \in B\}, \text{ and}$$

- *the **symmetric difference** set operator:*

$$A \oplus B = \{x | x \in A,\ x \notin B \text{ or } x \notin A,\ x \in B\}.$$

For example if $A = \{0, 1, 2\}$ and $B = \{1, 3\}$ we have

$$\begin{aligned} A \cup B &= \{0, 1, 2, 3\}, \\ A \cap B &= \{1\}, \text{ and} \\ A \oplus B &= \{0, 2, 3\}. \end{aligned}$$

Convention A.4 (Powerset)
For any set A we denote 2^A as the powerset or set of all subsets of A. For example, if $A = \{a_0, a_1, \ldots, a_{n-1}\}$, then

$$2^Q = \{\emptyset, \{q_0\}, \{q_1\}, \{q_0, q_1\}, \ldots, \{q_0, q_1, \ldots, q_{n-1}\}\}.$$

Definition A.5 (Alphabet)
An alphabet is a non-empty set of finite size. For our use it is the set of input symbols of a finite automaton.

Definition A.6 (Null Element)
The null element, denoted by ϵ, stands for no input symbol.

Convention A.7 (Kleene Closure)
Given a set Σ, the Kleene closure of Σ, is the set of all words

$$\Sigma^* = \{\sigma_1\sigma_2 \ldots \sigma_n\},\ \textit{with}\ \sigma_i \in \Sigma\ \textit{and}\ n \in \mathcal{Z}\}.$$

For example if $\Sigma = \{a, b\}$, then

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}.$$

Convention A.8 (Regular Expressions)
The language of a FA can be written as a regular expression, a regular expression is defined recursively as follows, where x, y are regular expressions,
$x = \sigma$ with $\sigma \in \Sigma$,
$x|y$ stands for either x or y,
x^+ stands for one or more instance of x,
(xy) stands for x followed by y,
$[x]$ stands for zero or one instances of x, and
x^* stands for the Kleene closure of the set $\{x\}$.

Definition A.9 (Output Finite Automata)
An output finite automaton is defined as a 6-tuple $M = (Q, \Sigma, \Theta, \delta, Q_0, \phi)$ where

- *Q is the finite non-empty set of states,*
- *Σ is the finite non-empty input alphabet,*
- *Θ is the finite non-empty **output alphabet**,*
- *$\delta : Q \times \Sigma \to 2^Q$ is the transition function,*
- *Q_0 is the set of initial states, and*
- *$\phi : Q \to \Theta$ is the output function.*

Corollary A.10 (Converting Output FAs to NFAs)
Any FA can be converted to an output FA as follows:

$$(Q, \Sigma, \delta, Q_0, F) \equiv (Q, \Sigma, \{0,1\}, \delta, Q_0, \phi_F)$$

where ϕ_F is defined as

$$\phi_F(q) = \begin{cases} 0 & , \quad q \notin F \\ 1 & , \quad q \in F \ , \end{cases}$$

where an output of 1 stands for an accept, while output of 0 stands for reject.

Definition A.11 (Inverses)
*Given a sequence $x = x_0 x_1 x_2 \ldots x_{n-1}$, its **inverse** is*

$$\overleftarrow{x} = x_{n-1} x_{n-2} \ldots x_0.$$

*Given a set of sequences $S = \{x, y, \ldots\}$ its **inverse** is*

$$\overleftarrow{S} = \{\overleftarrow{x}, \overleftarrow{y}, \ldots\}.$$

Definition A.12 (Isomorphism)
*Two DFAs $M_1 = (Q, \Sigma, \delta, q_0, F)$ and $M_2 = (Q', \Sigma, \delta', q_0', F')$ are **isomorphic** if there exists a function $f : Q \to Q'$ so that*

$$\begin{array}{rcl} f(q) & = & q', \\ f(\delta(q,\sigma)) & = & \delta'(q', \sigma), \\ q \in F & \Leftrightarrow & f(q) \in F', \ \textit{and} \\ f(q_0) & \Leftrightarrow & q_0'. \end{array}$$

where $\sigma \in \Sigma$.
It is easy to verify that $L(M_1) \equiv L(M_2)$.

Definition A.13
In an $m \times n$ matrix A with binary elements e_{ij} a row x is said to cover row y iff y has a 0 in every column where x has a 0. That is:

$$\begin{array}{lll} e_{xj} = 0 & \textit{if} & e_{yj} = 0, \ \textit{and} \\ e_{yj} = 1 & \textit{if} & e_{xj} = 1. \end{array}$$

□

Definition A.14 (Galois Field)
The Galois field $GF(2)$ contains the numbers 0 and 1 where addition and multiplication are defined as

$$\begin{array}{rcl} 1 + 1 & = & 0, \\ 1 + 0 & = & 1, \\ 0 \times 1 & = & 0, \\ 1 \times 1 & = & 1. \end{array}$$

□

Bibliography

[1] M. J. Atallah. *Handbook of Algorithms and Complexity.* CRC Press, 1998.

[2] J. Brzozowski. Derivatives of regular expressions. *Journal of the ACM 11*, pages 481–494, October 1964.

[3] J.M. Champarnaud and F. Coulon. NFA reduction algorithms by means of regular inequalities. *Developments in Language Theory 2003, Lecture Notes in Computer Science*, 2710:194–205, 2003.

[4] J. Daciuk, B.W. Watson, and R.E. Watson. Incremental construction of minimal acyclic finite-state automata and transducers. *L. Kartunnen (Ed.), Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 48–55, 1998.

[5] L.L. Dornhoff and F.E. Hohn. *Applied Modern Algebra.* MacMillan Publishing Co., Inc., New York, 1977.

[6] C. Hagenah and A. Muscholl. Computing ϵ-free NFA from regular expressions in $\mathcal{O}(n\log^2 n)$ time. *Theoretical Informatics and Applications*, 34:257–277, 2000.

[7] J. E. Hopcroft. An $n\log n$ algorithm for minimizing the states in a finite automaton. *Z. Kohavi (Ed.), The Theory of Machines and Computations*, pages 189–196, 1971.

[8] L. Ilie, G. Navarro, and S. Yu. On NFA reductions. *Theory is Forever, Lecture Notes in Computer Science*, 3113:112–124, 2004.

[9] L. Ilie and S. Yu. Algorithms for computing small NFAs. *K. Diks, W. Rytter (Eds.), Lecture Notes in Computer Science*, 2420:328–340, 2002.

[10] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM Journal on Computing*, 22(6):1171–1141, 1993.

[11] T. Kameda and P. Weiner. On the state minimization of nondeterministic automata. *IEEE Trans. Comput.*, C(19):617–627, 1970.

[12] J. Massey. Shift register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, pages 122–127, 1969.

[13] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[14] P.P. Chaidhuri, D. Chowdhury, S. Nandi and S. Chattopahyay. *Additive Cellular Automata: Theory and Applications, Vol. 1.* IEEE Computer Society Press, Los Alamitos, California, 1997.

[15] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, 1997.

[16] H. Tamm and E. Ukkonen. Bideterministic automata and minimal representations of regular languages. *Proceedings of The Eighth International Conference on Implementation and Application of Automata (CIAA 2003)*, pages 61–71, July 2003.

[17] L. van Zijl. *Generalized Nondeterminism and the Succinct Representation of Regular Languages.* PhD thesis, University of Stellenbosch, South Africa, March 1997.

[18] L. van Zijl. Random number generation with symmetric difference NFAs. In *Proceedings of the 6th International Conference on the Implementation and Application of Automata*, pages 263–273, Pretoria, South Africa, July 2001.

[19] Bruce W. Watson. A taxonomy of finite automata minimization algorithms. Technical report, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, the Netherlands, 1994.