HOW TO USE SORTING PROCEDURES TO MINIMIZE A DFA

by

Barbara Schubert

Department of Computer Science

Submitted in partial fulfilment

of the requirements for the degree of

Master of Science

Faculty of Graduate Studies

The University of Western Ontario

London, Ontario

December 1996

© Barbara Schubert 1997

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced with the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-21100-2

Canada

ABSTRACT

In this thesis we introduce a new idea, which can be used in the process of minimization of a deterministic finite automaton. Namely, we associate names with states of an automaton and we sort them. We give a new algorithm, its correctness proof, and a proof of its execution time bound. This algorithm has time complexity $O(n^2 \log n)$, where n is the number of states. In the thesis, we explain how to apply the new algorithm to an arbitrary partition of states, not only to the partition into final and nonfinal states. We present also two improved versions of the new algorithm (the second version and the third version). The second version of this algorithm which has time complexity $\Theta(n^2)$ can be considered as a direct improvement of Wood's algorithm [7]. Wood's algorithm has time complexity $\Theta(n^3)$. This algorithm checks whether pairs of states are distinguishable. It is improved by making better use of transitivity. Similarly some other algorithms which check if pairs of states are distinguishable can be improved using sorting procedures. The third version of the new algorithm which has time complexity $\Theta(n^2)$ is compared to the algorithm due to Hopcroft and Ullman [4] which has the same time complexity.

Acknowledgements

I would like to thank Dr. Helmut Jürgensen for his guidance and assistance.

I would like to thank my husband and my son, whose continuous encouragement and support made this thesis possible.

Finally, I would like to acknowledge the Natural Sciences and Engineering Research Council for their financial support of this research.

Table of Contents

Page

Certificate of Examination ii

Abstract iii

Acknowledgements iv

Table of Contents v

Chapter 1 – INTRODUCTION 1

Chapter 2 – BASIC NOTIONS 3

Chapter 3 – WOOD'S ALGORITHM 18

Chapter 4 – ALGORITHM DUE TO HOPCROFT AND ULLMAN 23

Chapter 5 – NEW ALGORITHMS 31

5.1. General Version 32

5.2. Second Version 42

5.3. Third Version 47

CHAPTER 6 – DIFFERENCE BETWEEN THE ALGORITHMS . 59

CHAPTER 7 – CONCLUDING REMARKS 62

7.1. Summary of Results 62

REFERENCES 63

VITA 64

Chapter 1

INTRODUCTION

The problem of minimization of a Deterministic Finite Automaton (DFA) is considered a fundamental computing science problem, and has been extensively studied. As this problem was studied, numerous algorithms were developed over the years. The asymptotically fastest known algorithm for minimization is Hopcroft's algorithm [3]. Its time complexity is $O(n \log n)$. A complete list of currently available algorithms for minimization can be found in [6]. Hopcroft's algorithm, while asymptotically fastest, is very complicated. In practice, less efficient, but less complex, algorithms are used – like Wood's algorithm [7], and an algorithm due to Hopcroft and Ullman [4].

We introduce a new idea, which can be used in minimization. Namely we associate names with states of an automaton and we sort them. This algorithm can be considered as a direct improvement of Wood's algorithm [7]. We will explain how to apply this algorithm to any arbitrary partition of states (not only to the partition into final and nonfinal states). The thesis is structured in the following way:

- The general definitions which deal with arbitrary partitions of states of an automaton are introduced.
- The algorithm due to Wood and the algorithm due to Hopcroft and Ullman are explained and examples for both algorithms are presented.

- The general version of a new algorithm which uses sorting is introduced and a proof of the correctness of the algorithm is presented. Then two special versions of the algorithm are explained. For the general version of the algorithm an upper bound on the run time is calculated. For the second and the third versions of the algorithm the least upper bound on the run time is calculated.
- It is explained when sorting is an improvement. The second version of the algorithm is compared to Wood's algorithm. The third version of the algorithm is compared to the algorithm due to Hopcroft and Ullman.

In this thesis we present new algorithms for minimization. Specifically we achieve the following:

- Show how sorting of states can be used to minimize an automaton.
- Present three versions of a new algorithm.
- Explain when sorting of states can be more efficient by comparing two versions of the new algorithm to two known algorithms.
- Present a new algorithm in a way that can be used to solve directly problems. where states are partitioned into more than two blocks (like problems presented in [1]).

CHAPTER 2

BASIC NOTIONS

In this section we give the basic definitions concerning automata and establish the relevant notation. Instead of the usual definition of an automaton as an acceptor we define an automaton as a diagnoser in the sense of [1]. As an acceptor an automaton has two types of states. final and nonfinal. As a diagnoser. it may have more then these two types of states. Accordingly. in a diagnoser input words are classified by the state reached. Specifically in an acceptor input words are classified as accepted or not accepted. Diagnosers model. for instance. certain aspects of the process of circuit testing.

We only give the definitions absolutely needed. For further notions and properties of automata we refer to [2].

Definition 2.1 Let S be a set. A *partition* Π of S is a set of pairwise disjoint subsets of S. the union of which is S.

For technical reasons we allow a partition to contain the empty set as an element. Sometimes we need to exclude it explicitly. Thus if Π is a partition of S. then the *partition* $\hat{\Pi}$ is defined as follows:

$$\hat{\Pi} = \Pi \setminus \{\emptyset\}.$$

Definition 2.2 A *deterministic finite automaton (DFA)* M is specified by a quintuple $M = (Q, \Sigma, \delta, s, B)$ where

Q is the alphabet of *state* symbols;

Σ is the alphabet of *input* symbols;

$\delta : Q \times \Sigma \to Q$ is the *transition function*;

$s \in Q$ is the *start state*;

$B = (B_1, B_2, \dots, B_r)$ is a partition of the states of the automaton.

As usual, we extend the transition function δ to a function of $Q \times \Sigma^*$ into Q by

$$\delta(q, w) = \begin{cases} q, & \text{if } w = \lambda, \\ \delta(\delta(q, x), w'), & \text{if } w = xw' \text{ with } x \in \Sigma, w' \in \Sigma^*, \end{cases}$$

for all $q \in Q$ and $w \in \Sigma^*$, where λ denotes the empty word. Instead of $\delta(q, w)$ we write qw. The partition of states of an automaton allows one to classify the input words. An input word $w \in \Sigma^*$ is of type B_i if $sw \in B_i$. Let ϱ_B be the equivalence relation on Q defined by B. For $q \in Q$ we write $[q]_B$ to denote the ϱ_B-class of q.

When $|B| = 2$, a DFA in the sense of Definition 2.2 is the same as a DFA in the usual sense with, for example, B_1 the set of final states and $B_2 = Q \backslash B_1$. The general definition is illustrated by an example below.

Definition 2.3 Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA. The *language classification* γ_M defined by M is the mapping $\gamma_M : B \to 2^{\Sigma^*}$ such that

$$\gamma_M(B_i) = \{w \in \Sigma^* \mid sw \in B_i\}$$

If $sv, sw \in B_i$ for some $B_i \in B$, we say that v and w are *equivalent* and we write $v \equiv^M w$.

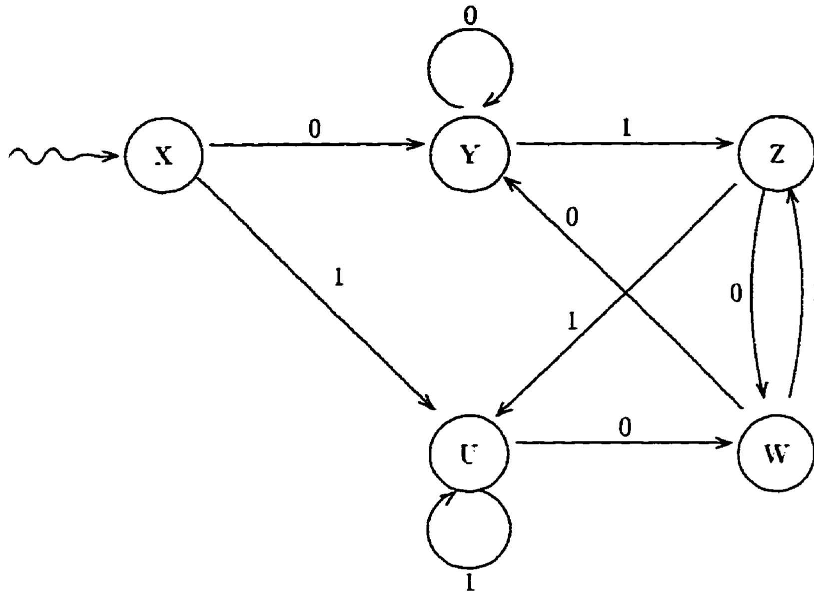

Figure 2.1. Deterministic finite automaton M_0.

Definition 2.3 allows us to conclude that the following statement is true: if $[sw]_B = [sv]_B$ then $v, w \in \gamma_M(B_i)$ for some $B_i \in B$, that is, $v \equiv^M w$.

Whenever we have a partition into two sets of final and nonfinal states, we can classify the input words into the words accepted and not accepted by the automaton. An example of the language classification for a different partition is presented in the following example:

Example 2.1 Consider the automaton from Figure 2.1. Suppose that we have the following partition of the states in this automaton: $B_1 = \{X\}$, $B_2 = \{Y\}$, $B_3 = \{U\}$, and $B_4 = \{W, Z\}$. This partition allows us to classify input words into four sets: $\{\lambda\}$ corresponding to the block B_1, $\{0\} \cup \Sigma^*00$ corresponding to the block B_2, $\{1\} \cup \Sigma^*11$ corresponding to the block B_3, and $\Sigma^*01 \cup \Sigma^*10$ corresponding to

the block B_4.

Lemma 2.1 *Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA and $|B| = r$. Then the family*

$$\hat{\Pi}^M = \{\gamma_M(B_i) \mid 1 \leq i \leq r, \gamma_M(B_i) \neq \emptyset\}$$

is a partition of Σ^*.

Proof: To prove that $\hat{\Pi}^M$ is a partition, we have to prove that for every $w \in \Sigma^*$ there exists exactly one $B_i \in B$ such that $w \in \gamma_M(B_i)$. As the automaton M is complete and deterministic, for every $w \in \Sigma^*$ there exists exactly one $B_i \in B$ such that $sw \in B_i$. Thus $w \in \gamma_M(B_i)$.

Definition 2.4 Let $M = (Q, \Sigma, \delta, s, B)$ and $M' = (Q', \Sigma, \delta', s', B')$ be DFAs. Then M and M' are equivalent if $\hat{\Pi}^M = \hat{\Pi}^{M'}$.

Using the notation from Definition 2.3 we can state that M and M' are equivalent if the following statement is true:

for every $v, w \in \Sigma^*$ $w \equiv^M v$ if and only if $w \equiv^{M'} v$.

Definition 2.5 Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA. We define the automaton $\bar{M}$ in the following way: $\bar{M} = (\bar{Q}, \Sigma, \bar{\delta}, s, \bar{B})$ where

$\bar{Q} = \{q \mid q = sw$ for some $w \in \Sigma^*\}$,

$\bar{B} = \{\bar{B}_i \mid \exists q \in \bar{Q}$ such that $q \in \bar{B}_i\}$, where $\bar{B}_i = B_i \cap \bar{Q}$,

and $\bar{\delta} : \bar{Q} \times \Sigma \rightarrow \bar{Q}$.

We can state that $\bar{B} = \{\bar{B}_i \mid \gamma_M(\bar{B}_i \neq \emptyset\}$. Thus $\bar{B}$ contains blocks with the states reachable from the initial state s. Every $\bar{B}_i \in \bar{B}$ contains at least one reachable state. $B \setminus \bar{B}$ contains either empty blocks or blocks which have only states that

cannot be reached from the initial state. Notice that $\gamma_M(B_j)$ for $B_j \in B \setminus \bar{B}$ is the empty set. Thus the automaton $\bar{M}$ consists only from reachable states.

Lemma 2.2 *Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA. Then M and $\bar{M}$ are equivalent and $|Q| \geq |\bar{Q}|$.*

Proof: Assume that $v \equiv^M w$ for some $v, w \in \Sigma^*$. Then $v, w \in \gamma_M(B_i)$ for some $B_i \in B$. As $\gamma_M(B_i) \neq \emptyset$ we have $B_i \in \bar{B}$. Thus $v \equiv^{\bar{M}} w$.
Let $v \equiv^{\bar{M}} w$ for some $v, w \in \Sigma^*$. Then $v, w \in \gamma_M(\bar{B}_j)$ for some $\bar{B}_j \in \bar{B}$. As $\bar{B} \subseteq B$ we have $\bar{B}_j \in B$. Thus $v \equiv^M w$. If M has unreachable states then $|Q| > |\tilde{Q}|$. Otherwise $|Q| = |\tilde{Q}|$.

Thus an automaton M with unreachable states is equivalent to the subautomaton of M which contains all states of M except unreachable states. The automaton $\bar{M}$ has at most the same number of states as M. Whenever we want to decide the equivalence of two automata M and M', we can just decide the equivalence of $\bar{M}$ and $\bar{M}'$.

Definition 2.6 Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA. A congruence $\sim$ on M is an equivalence relation on Q such that $\forall q, q' \in Q, \forall a \in \Sigma^*$

1. if $q \sim q'$ then $qa \sim q'a$
2. if $q \sim q'$ and $q \in [q']_B$ then $qa \in [q'a]_B$

The second condition from Definition 2.6 can be rewritten in the following way: if $q \sim q'$ and $[q]_B = [q']_B$ then $[qa]_B = [q'a]_B$. The first condition is equivalent to the following statement: if $q \sim q'$ then $qw \sim q'w$ for every $w \in \Sigma^*$. The correctness of this statement can be proved using induction.

Definition 2.7 Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA. Two states p and q of M are said to be *equivalent* if, for every $w \in \Sigma^*$, $[qw]_B = [pw]_B$. We write $p \equiv q$ in

this case. The states p and q are ***distinguishable*** if there exists $w \in \Sigma^*$ such that $[pw]_B \neq [qw]_B$: then the word w ***distinguishes*** p from q.

Lemma 2.3 *The relation $\equiv$ is an automaton congruence on M and contained in ϱ_B. Moreover, if $\sim$ is any automaton congruence on M which is contained in ϱ_B then $\sim \subseteq \equiv$.*

Proof: To prove that the relation $\equiv$ is an automaton congruence we have to show first that whenever $p \equiv q$ then $pa \equiv qa$ for all $a \in \Sigma$. Let $q, p \in Q, a \in \Sigma$. By Definition 2.7

$$pa \equiv qa \Leftrightarrow \forall w \in \Sigma^* \; [paw]_B = [qaw]_B$$

Thus we have to prove that for all $w \in \Sigma^*, [paw]_B = [qaw]_B$. As $p \equiv q$, $[pw']_B = [qw']_B$ for all $w' \in \Sigma^*$. So also for every $w \in \Sigma^*$ such that $w' = aw$ we get $[paw]_B = [qaw]_B$ as required. Now suppose that $p \equiv q$ and $p \in [q]_B$. Then $pa \equiv qa$ and so $[pa]_B = [qa]_B$ by the definition of $\equiv$.
We now prove that $\equiv$ is contained in ϱ_B. By the definition, if $p \equiv q$, then $[pw]_B = [qw]_B$ for all $w \in \Sigma^*$. Let $w = \lambda$. Thus $[p]_B = [q]_B$.
To prove that $\sim \subseteq \equiv$ let $p, q \in Q$ be states for which $p \sim q$. As $\sim$ is contained in ϱ_B we have $[p]_B = [q]_B$. As this is a congruence relation $pa \sim qa$ for every $a \in \Sigma$ and so $[pa]_B = [qa]_B$. By induction we conclude that for every $w \in \Sigma^*$ $[pw]_B = [qw]_B$ and so $p \equiv q$.

Definition 2.8 Let $M = (Q, \Sigma, \delta, s, B)$ and $M' = (Q', \Sigma, \delta', s', B')$ be DFAs. Then we say that the system $\Phi = (\kappa_\Phi, \mu_\Phi)$ consisting of the mappings $\kappa_\Phi : Q \to Q'$ where $\kappa_\Phi(s) = s'$, and $\mu_\Phi : B \to B'$ is a *homomorphism* of M into M' if for arbitrary $p \in Q$ and $w \in \Sigma^*$,

$$\kappa_\Phi(pw) \in \mu_\Phi([pw]_B)$$

and

$$\kappa_\Phi(pw) = \kappa_\Phi(p)w.$$

It is obvious that the homomorphic image of M is a subautomaton of M'. If κ_Φ and μ_Φ are onto mappings then the system Φ is called surjective and M' is called the homomorphic image of M. If additionally κ_Φ and μ_Φ are one-to-one mappings then the system Φ is an isomorphism and the automata M and M' are said to be isomorphic.

Definition 2.9 Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA and $\sim$ be a congruence on M contained in ϱ_B and $B = \{B_1, B_2, \ldots, B_r\}$. We define the factor automaton $M/_\sim = (Q/_\sim, \Sigma, \delta/_\sim, s_\sim, B/_\sim)$ with respect to that congruence where:

$$q_\sim = \{p \in Q \mid p \sim q\}$$
$$B_i/_\sim = \{q_\sim \mid q \in B_i\}$$
$$B/_\sim = (B_1/_\sim, B_2/_\sim, \ldots, B_r/_\sim\}$$
$$\delta/_\sim(q_\sim, a) = (qa)_\sim$$

Notice that as $\sim$ is contained in ϱ_B we have that $B/_\sim$ consists of disjoint blocks. Whenever B_i is the empty set, the block $B_i/_\sim$ is also the empty set. Obviously $|B| = |B/_\sim|$. As far as the behavior of $M/_\sim$ is concerned, we do not treat $p_\sim$ as a set, but as a state. Hence we write $p_\sim a$ to denote $\delta/_\sim(p_\sim, w)$. Thus $p_\sim a = (pa)_\sim$ whereas in the form of sets one could only state that $p_\sim a \subseteq (pa)_\sim$.

Lemma 2.4 *Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA and $\sim$ be a congruence on M contained in ϱ_B. Let $M/_\sim = (Q/_\sim, \Sigma, \delta/_\sim, s_\sim, B/_\sim)$ be the factor automaton. Then for*

all $w \in \Sigma^*$ and $p \in Q$ we have $p_\sim w = (pw)_\sim$.

Proof: By Definition 2.9 for all $a \in \Sigma$ we have that $p_\sim a = (pa)_\sim$. Thus assume that $p_\sim w' = (pw')_\sim$ for some $w' \in \Sigma^*$. Then $p_\sim w'a = (pw')_\sim a$ and by Definition 2.9 $(pw')_\sim a = (pw'a)_\sim$, that is, $p_\sim w'a = (pw'a)_\sim$.

Theorem 2.1 *Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA and $\sim$ be a congruence on M contained in ϱ_B. Let $M/_\sim = (Q/_\sim, \Sigma, \delta/_\sim, s_\sim, B/_\sim)$ be the factor automaton. Then $\Phi_\sim = (\kappa_\sim, \mu_\sim)$ with $\kappa_\sim(q) = q_\sim$ and $\mu_\sim(B_i) = B_i/_\sim$, for $q \in Q$ and $B_i \in B$, is a homomorphism of M onto $M/_\sim$.*

Proof: To prove that the map $\Phi_\sim$ is a homomorphism we have to show that, for arbitrary $p \in Q$ and $w \in \Sigma^*$, $\kappa_\sim(pw) = \kappa_\sim(p)w$ and $\kappa_\sim(pw) \in \mu_\sim([pw]_B)$. Let $p \in Q$ and $w \in \Sigma^*$. As for every $q \in Q$ we have $\kappa_\sim(pw) = (pw)_\sim$ thus by Lemma 2.4 $\kappa_\sim(pw) = (pw)_\sim = (p_\sim)w = \kappa_\sim(p)w$. Also $\mu_\sim([pw]_B) = \mu_\sim(B_i) = B_i/_\sim$ where $pw \in B_i$. As $\kappa_\sim(pw) = (pw)_\sim$ and $(pw)_\sim \in B_i/_\sim$ we have $\kappa_\sim(pw) \in B_i/_\sim$. To prove that the map $\Phi_\sim$ is onto, let $q_\sim \in Q/_\sim$. Then, for $q \in Q$, $\kappa_\sim(q) = q_\sim$.

Notice that by Lemma 2.3, given automata M, $M/_\sim$, and $M/_\equiv$, the following relation is always true: $|Q| \geq |Q_\sim| \geq |Q_\equiv|$.

Definition 2.10 Let $M = (Q, \Sigma, \delta, s, B)$ and $M' = (Q', \Sigma, \delta', s', B')$ be DFAs. A *block re-assignment* is a function $\zeta : B \to B'$ satisfying

1. for all $w \in \Sigma^*$, $\zeta([sw]_B) = [s'w]_{B'}$,
2. the restriction of ζ to $\bar{B}$, $\bar{\zeta} : \bar{B} \to \bar{B}'$, is injective.

Theorem 2.2 *Let $M = (Q, \Sigma, \delta, s, B)$ and $M' = (Q', \Sigma, \delta', s', B')$ be DFAs. M and M' are equivalent if and only if there exists a block re-assignment $\zeta : B \to B'$.*

Proof: Suppose that M and M' are equivalent. Define ζ in the following way:

$$\zeta(B_i) = \begin{cases} [s'w]_{B'}, & \text{if } B_i = [sw]_B, \\ B_j \in B' & \text{if } B_i \in B \setminus \bar{B} \end{cases}$$

where B_j is arbitrary but fixed.

As every element in $\bar{B}$ can be written as $[sw]_B$ for some $w \in \Sigma^*$, we see that ζ is defined for every element in the domain. To prove that ζ is well defined suppose that that for some $B_j = [sv]_B$ and $B_j = [sw]_B$, we have $[sv]_B = [sw]_B$. Thus $v \equiv^M w$. As M and M' equivalent we have $v \equiv^{M'} w$ and so $[s'v]_{B'} = [s'w]_{B'} = \zeta(B_j)$. This proves that the map ζ is well defined.

By the definition of ζ, the first condition in Definition 2.10, that $\zeta([sw]_B) = [s'w]_{B'}$ is satisfied.

We have to prove that the image of $\bar{\zeta}$ is a subset of $\bar{B}'$. Let $[sw]_B \in \bar{B}$. Then $\bar{\zeta}([sw]_B) = [s'w]_{B'}$ by the definition. As $[s'w]_{B'} \in \bar{B}'$ we have proved that the image of $\bar{\zeta}$ is a subset of $\bar{B}'$.

To prove that the map $\bar{\zeta}$ is one-to-one let $\bar{\zeta}(B_i) = \bar{\zeta}(B_j)$ where $B_i, B_j \in \bar{B}$. We have to prove that $B_i = B_j$. The image of $\bar{\zeta}$ is in $\bar{B}'$. Thus $\bar{\zeta}(B_i) = [s'w]_{B'}$ and $\bar{\zeta}(B_j) = [s'v]_{B'}$ for some $v, w \in \Sigma^*$. Thus $w \equiv^{M'} v$. As M and M' are equivalent $w \equiv^M v$. Thus $[sw]_B = [sv]_B$. This proves that $\bar{\zeta}$ is one-to-one.

Now suppose that there exists ζ with the required properties. We have to prove that M and M' are equivalent. Let $w \equiv^M v$ for some $w, v \in \Sigma^*$. Thus $[sw]_B = [sv]_B$. But then $\zeta([sw]_B) = \zeta([sv]_B)$ as ζ is well defined and so $[s'w]_{B'} = [s'v]_{B'}$. Thus $w \equiv^{M'} w'$. Let $w \equiv^{M'} v$. Then $[s'w]_{B'} = [s'v]_{B'}$. As $\bar{\zeta}$ is one-to-one $[sw]_B = [sv]_B$. Thus $w \equiv^M v$.

Corollary 2.1 *Let $M = (Q, \Sigma, \delta, s, B)$ and $M' = (Q', \Sigma, \delta', s', B')$ be DFAs. Let $\zeta : B \to B'$ be a block re-assignment. Then the map $\bar{\zeta}$ is onto.*

Proof: Let $[s'w]_{B'} \in \bar{B}'$. By the definition of $\bar{\zeta}$, for $[sw]_B \in \bar{B}$ we have $\bar{\zeta}([sw]_B) =$

$[s'w]_{B'}$ (sw is defined as the automaton M is complete). This proves that the map $\bar{\zeta}$ is onto.

Let $M = (Q, \Sigma, \delta, s, B)$ and $M' = (Q', \Sigma, \delta', s', B')$ be DFAs. If there exists a block re-assignment ζ, we say that M and M' are ζ-*equivalent.*

Definition 2.11 Let $M = (Q, \Sigma, \delta, s, B)$ and $M' = (Q', \Sigma, \delta', s', B')$ be ζ-equivalent DFAs for some block re-assignment $\zeta : B \rightarrow B'$. Two reachable states $p \in Q$ and $p' \in Q'$ are said to be ζ-*equivalent* if for every $w \in \Sigma^*$ we have $\zeta([pw]_B) = [p'w]_{B'}$. We write $p \equiv^{\zeta} p'$ in this case.

Two states p and q are equivalent if, applying any input word w, the resulting states belong to the same block B_i. Whenever we consider two ζ-equivalent automata M and M', two states $p \in Q$ and $p' \in Q'$ are considered to be equivalent if, whenever a state pw is in a block B_i, then the state $p'w$ is in the block $\zeta(B_i)$.

As the map $\bar{\zeta}$ is one-to-one and onto, two automata without unreachable states are equivalent if and only if they are ζ-equivalent or ζ^{-1}-equivalent. Similarly, if two reachable states are ζ-equivalent, they are also ζ^{-1}-equivalent.

Lemma 2.5 *Let $M = (Q, \Sigma, \delta, s, B)$, and $M' = (Q', \Sigma, \delta', s', B')$ be DFAs which are ζ-equivalent for some block re-assignment $\zeta : B \rightarrow B'$. For reachable states $p \in Q$ and $p', q' \in Q'$, if $p \equiv^{\zeta} p'$ and $p \equiv^{\zeta} q'$ then $p' \equiv q'$.*

Proof: As $p \equiv^{\zeta} p'$, then by Definition 2.11 $\zeta([pw]_B) = [p'w]_{B'}$ for every $w \in \Sigma^*$. Similarly if $p \equiv^{\zeta} q'$, then $\zeta([pw]_B) = [q'w]_{B'}$. Thus $\zeta([pw]_B) = [p'w]_{B'} = [q'w]_{B'}$ for every $w \in \Sigma^*$ which implies that $p' \equiv q'$.

Lemma 2.6 *Let $M = (Q, \Sigma, \delta, s, B)$, and $M' = (Q', \Sigma, \delta', s', B')$ be ζ-equivalent DFAs. For reachable states $p, q \in Q$ and $q' \in Q'$, if $p \equiv q$ and $q \equiv^{\zeta} q'$ then $p \equiv^{\zeta} q'$.*

Proof: As $p \equiv q$ then $[pw]_B = [qw]_B$ for every $w \in \Sigma^*$. As $q \equiv^\zeta q'$. then by Definition 2.11 $\zeta([qw]_B) = [p'w]_{B'}$ for every $w \in \Sigma^*$. Thus $\zeta([pw]_B) = [q'w]_{B'}$ for every $w \in \Sigma^*$. which implies that $p \equiv^\zeta q'$.

Lemma 2.7 *Let $M = (Q.\Sigma.\delta.s.B)$. and $M' = (Q'.\Sigma.\delta'.s'.B')$ be DFAs without unreachable states which are ζ-equivalent. Then for every $p \in Q$ there exists $p' \in Q'$ such that $p \equiv^\zeta p'$. Similarly for every $p' \in Q'$ there exists $p \in Q$ such that $p \equiv^\zeta p'$.*

Proof: Let $p \in Q$. As every state in M is reachable there exists $w \in \Sigma^*$ such that $p = sw$. By the definition of ζ we have $\zeta([sw]_B) = [s'w]_{B'}$. Let $p' = s'w$. Also by the definition of ζ we have that. for every $w' \in \Sigma^*$. $\zeta([sww']_B) = [s'ww']_{B'}$. Thus for every $w' \in \Sigma^*$ we have $\zeta([pw']_B) = [p'w']_{B'}$ and so $p \equiv^\zeta p'$.
Let $p' \in Q'$. As every state in M' is reachable there exists $w \in \Sigma^*$ such that $p' = s'w$. Notice that $p' \in \bar{B}'_i$ for some $\bar{B}'_i \in \bar{B}'$. As the function $\bar{\zeta}$ is onto. there exists $p = sw \in Q$ such that $\zeta([sw]_B) = [s'w]_{B'}$. Also by the definition of ζ we have that. for every $w' \in \Sigma^*$. $\zeta([sww']_B) = [s'ww']_{B'}$. Thus for every $w' \in \Sigma^*$ we have $\zeta([pw']_B) = [p'w']_{B'}$ and so $p \equiv^\zeta p'$.

Definition 2.12 Let $M = (Q.\Sigma.\delta.s.B)$ be a DFA. The automaton M is minimal if for all automata $M' = (Q'.\Sigma'.\delta'.s.B')$ equivalent to M. $|Q| \leq |Q'|$.

The known algorithms for minimization of a DFA are usually explained using the partition of states into the two sets of final and nonfinal states. so $B = (F.Q\backslash F)$. For the partition into final and nonfinal states two automata are equivalent if and only if they accept the same language. The minimal automaton M' constructed from the automaton M should accept the same language as M and have the minimal number of states.

Thus one should be careful applying the above definitions and theorems to a

specific example. It can be the case that only one mapping is permitted and so the definitions of the equivalence should be modified. For example given automata $M = (Q, \Sigma, \delta, s, F)$ and $M' = (Q', \Sigma, \delta', s', F')$, where F and F' are the sets of final states, the only permitted map is μ where $\mu(F) = \mu(F')$ and $\mu(Q \setminus F) = Q' \setminus F'$. However, as far as minimization is concerned, this restriction is not relevant: if L is the language accepted by M then the minimal acceptors for L and $\Sigma^* \setminus L$ are isomorphic.

Theorem 2.3 *Let $M = (Q, \Sigma, \delta, s, B)$, and $M' = (Q', \Sigma, \delta', s', B')$ be DFAs with $B = \bar{B}$ and $B' = \bar{B}'$ (thus there are no unreachable states and there are no empty blocks). If M and M' are equivalent then there is a DFA M'' and there are surjective homomorphisms Φ and Φ', such that $\Phi : M \to M''$ and $\Phi' : M' \to M''$.*

Proof: Let $M'' = M/_{\equiv}$ and $\Phi = \Phi_{\equiv}$ as defined in Theorem 2.1. It was proved in Theorem 2.1 that $\Phi_{\equiv} : M \to M/_{\equiv}$ is a surjective homomorphism. Notice that, as M and M' are equivalent, they are ζ-equivalent for some block re-assignment $\zeta : B \to B'$. Also notice that $\zeta = \bar{\zeta}$. As $B = \bar{B}$ and $B' = \bar{B}'$ we have $\bar{\zeta} = \zeta$.
Define $\Phi' : M' \to M/_{\equiv}$ in the following way:

$$\kappa'(q') = q_{\equiv} \text{ where } q' \equiv^{\zeta} q$$

$$\mu'(B'_i) = \mu(\bar{\zeta}(B'_i))$$

We have to prove that Φ' a surjective homomorphism.

1. First prove that Φ' is a mapping.

   Let $q' \in Q'$. As M and M' are equivalent, by Lemma 2.7 there exists $q \in Q$ such that $q' \equiv^{\zeta} q$. Thus $\kappa'(q') = q_{\equiv}$. Let $B'_i \in B'$. As ζ is defined on its domain there exists $B_j \in B$ such that $\zeta(B'_i) = B_j$. Thus as μ is defined on its domain then μ' is also.

   Suppose $q' \equiv^{\zeta} q$ and $q' \equiv^{\zeta} p$. By Lemma 2.5 $p \equiv q$. Thus $p_{\equiv} = q_{\equiv}$. Hence κ' is well defined.

This proves that Φ' is well defined.

2. We have to prove that Φ' is a homomorphism.

   We first prove that $\kappa'(p'w) = \kappa'(p')w$ for every $p' \in Q'$ and $w \in \Sigma^*$. By definition $\kappa'(p'w) = q_\equiv$ where $p'w \equiv^\zeta q$. By Lemma 2.7 there exists $p \in Q$ such that $p' \equiv^\zeta p$. By the definition of $\equiv^\zeta$, $p'w \equiv^\zeta pw$. By Lemma 2.5 $pw \equiv q$ and so $q_\equiv = (pw)_\equiv$. Finally, by Lemma 2.4 $(pw)_\equiv = p_\equiv w = \kappa'(p')v$.

   We have to prove also that $\kappa'(p'w) \in \mu'([p'w]_{B'})$ for every $p' \in Q'$ and $w \in \Sigma^*$. By Lemma 2.7 there exists $q \in Q$ such that $p' \equiv^\zeta q$. By the definition of $\equiv^\zeta$ we have $\zeta([p'w]_{B'} = [qw]_B$. Thus $\mu(\zeta([p'w]_{B'}) = \mu([qw]_B) = B_i/_\equiv$ where $(qw)_\equiv \in B_i/_\equiv$.

3. To prove that the map Φ' is onto let $q_\equiv \in Q/_\equiv$. By Lemma 2.7 there exists $q' \in Q'$ such that $q' \equiv^\zeta q$. Thus $\kappa'(q') = q_\sim$. As μ and ζ are onto, then the map μ' is also onto.

Corollary 2.2 *For every DFA $M = (Q, \Sigma, \delta, s, B)$ containing only reachable states there is a minimal DFA $M' = (Q', \Sigma, \delta', s', B')$ equivalent to M, which is unique up to isomorphism. Moreover M' is isomorphic with $M/_\equiv$.*

Proof: First we will prove that the automaton $M_\equiv$ is minimal. By contradiction assume that there exists an automaton M'' such that M'' and $M_\equiv$ are equivalent and $|Q''| < |Q/_\equiv|$. As the automata M'' and $M_\equiv$ are equivalent they are ζ-equivalent for some $\zeta : B/_\equiv \to B''$. By Lemma 2.7 for every $z'' \in Q''$ there exists $z_\equiv \in Q$ such that $z'' \equiv^\zeta z_\equiv$ and for every $z_\equiv \in Q$ there exists a corresponding $z'' \in Q''$. As $|Q''| < |Q/_\equiv|$ there exist distinct states $p_\equiv, q_\equiv \in Q/_\equiv$ and a state $p'' \in Q'$ such that $p_\equiv \equiv^\zeta p''$ and $q_\equiv \equiv^\zeta p''$. But then by Lemma 2.5 we have $q_\equiv \equiv p_\equiv$ and so we should have $p_\equiv = q_\equiv$ a contradiction.

To prove that the minimal automaton is unique up to isomorphism suppose that we have a minimal automaton $M' \neq M_\equiv$ which is equivalent to M. Consider the maps

Φ and Φ' defined in Theorem 2.3. The map $\Phi : M_{\equiv} \to M_{\equiv}$ is the identity map. By Theorem 2.3 the map $\Phi : M' \to M_{\equiv}$ is onto. As the cardinality of the minimal automata is the same ($|Q| = |Q'|$) we can conclude that the map $\Phi' \circ \Phi = \Phi'$ is an isomorphism.

Corollary 2.2 allows us to conclude that one of the ways to construct the minimal automaton is to search for $\equiv$. the largest congruence contained in ϱ_B. The next definitions will be used directly in the construction of the algorithm for minimization.

Definition 2.13 Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA. Given an integer $k \geq 0$. we say two distinct states $p, q \in Q$ are ***k-distinguishable*** if there is a word $w \in \Sigma^*$. $|w| \leq k$. which ***distinguishes*** p from q. that is $[pw]_B \neq [qw]_B$. If p and q are k-distinguishable we write $p \not\equiv^k q$. If there is no such word. then we say that p and q are ***k-indistinguishable*** and we write $p \equiv^k q$.

Theorem 2.4 *Let $M = (Q, \Sigma, \delta, s, B)$ be a DFA. For two distinct states $p, q \in Q$ with $p \equiv^k q$. $p \equiv^{k+1} q$ if and only if. for all $a \in \Sigma$. $pa \equiv^k qa$.*

Proof: By $|w|$ denote the length of the word w. Suppose that $p \equiv^{k+1} q$ for some $p, q \in Q$ and $k \geq 0$. Then $[pw]_B = [qw]_B$ for every $w \in \Sigma^*$ such that $|w| \leq k+1$ (thus $k+1 \geq 1$). As $0 \leq |a| \leq 1$ for all $a \in \Sigma$. we have $pa \equiv^k qa$.
Suppose that for all $a \in \Sigma$. $pa \equiv^k qa$. As $pa \equiv^k qa$ no word $w \in \Sigma^*$ such that $|w| \leq k$ distinguishes pa and qa. Thus $[paw]_B = [qaw]_w$. As every $w' \in \Sigma^*$ such that $|w'| \leq k+1$ can be expressed as $w' = aw$. we have $p \equiv^{k+1} q$.

Clearly two states are distinguishable if they are k-distinguishable. for some $k \geq 0$. If we cannot distinguish p and q with words of length at most $k+1$. for some $k \geq 0$. then we cannot distinguish them with words of length at most k. Thus. $p \equiv^{k+1} q$ implies $p \equiv^k q$. for all $k \geq 0$. that is. $\equiv^{k+1} \subseteq \equiv^k$. We say $\equiv^{k+1}$ is a *refinement* of

$\equiv^k$. Similarly, we obtain $\equiv \subseteq \equiv^k$, for all $k \geq 0$.

There are various definitions around for Ω. For the discussion about options see [6]. In this thesis the following definition is used:

Definition 2.14

$O(f(n)) = \{g \mid \exists_{c>0}\exists_{N>0}\forall_{n>N}[g(n) \leq cf(n)]\}$

$\Omega(f(n)) = \{g \mid \exists_{c>0}\forall_{N>0}\exists_{n>N}[g(n) \geq cf(n)]\}$

$\Theta(f(n)) = \{g \mid \exists_{c>0}\exists_{N>0}\forall_{n>N}[g(n) \leq cf(n)] \textit{ and } \exists_{c>0}\forall_{N>0}\exists_{n>N}[g(n) \geq cf(n)]\}$

Chapter 3

WOOD'S ALGORITHM

In this section we present Wood's algorithm [7] for the minimization of complete DFAs. This algorithm is illustrated by an example, in which the *tabular method* is used to construct a minimal DFA.

Algorithm 3.1

Input: A complete DFA with set of states Q, all states reachable, set of input symbols Σ, transitions defined for all states and inputs, start state s, and the partition of states B.

Output: A minimal DFA M' equivalent to the DFA M.

Method:

1. Construct an initial partition Π^k of the set of states for $k = 0$. This partition has at most $|B|$ blocks of states.
2. Let $k := k + 1$ and apply the following procedure to construct a new partition Π^k. Initially let $\Pi^k := \Pi^{k-1}$.

   For each block G of Π^k, for which $|G| \geq 2$ **do**

   begin

   partition G into subblocks such that two states s and t

   of G are in the same subblock if and only if for all

   input symbols a, the states sa and ta are in the same block of Π^{k-1}

/* at worst. a state will be in a subblock by itself */

replace G in Π^k by the set of all subblocks formed

end.

3. **If** $\Pi^{k-1} = \Pi^k$. let $\Pi := \Pi^k$

   else repeat from step (2).

4. Construct the minimal automaton using the partition Π.

Wood's algorithm has time complexity $O(n^3)$.

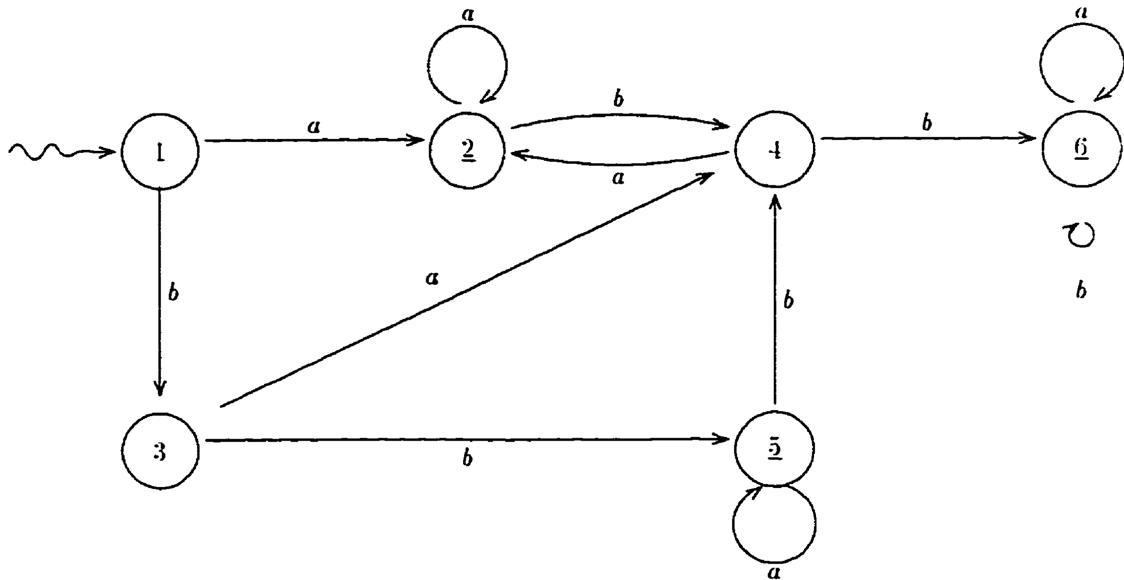

Figure 3.1. Deterministic finite automaton M_1 with the partition of states into two blocks: $B_1 = \{1,3,4\}$ and $B_2 = \{\underline{2},\underline{5},\underline{6}\}$.

Let Π^k be the partition of states at step k. The initial partition of states. Π^0. is defined as B. as two states are 0-indistinguishable if and only if they are in the same block of B. Thus $\equiv^0 = \varrho_B$. There are $r = |B|$ equivalence classes formed by $\equiv^0$. If all states are equivalent to each other. and so $|\Pi^0| = 1$ we can terminate the algorithm and the minimal automaton will have only one state. At step 2 we calculate new equivalence classes and so a new partition of states Π^1. and compare it with the partition Π^0. Then we are using Theorem 2.4. to construct new partitions

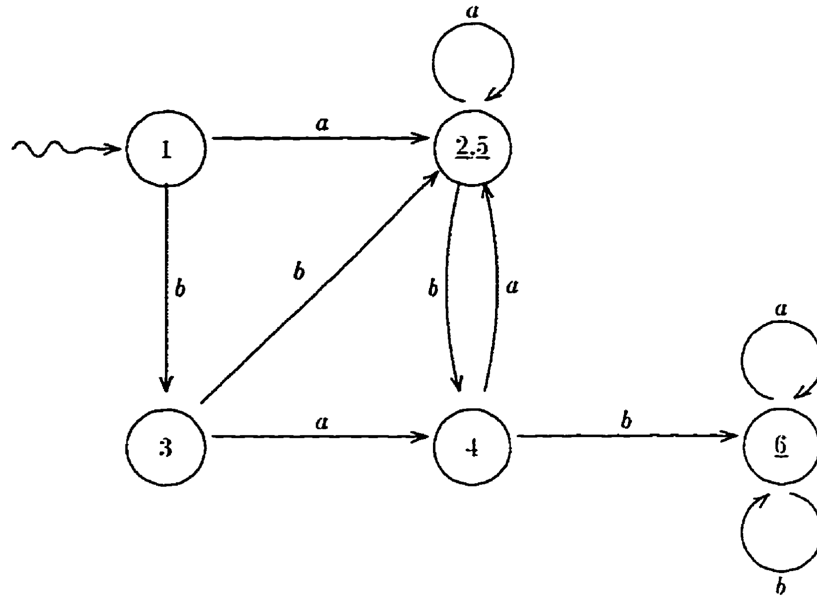

Figure 3.2. Minimum state finite automaton M_1'. This automaton is equivalent to the automaton M_1.

of states into equivalence classes (Π^2, then Π^3, and so on). Thus we construct $\equiv$ by refining $\equiv^0$ to give $\equiv^1$, $\equiv^1$ to give $\equiv^2$ and so on. This continues until $\equiv^{k-1} = \equiv^k$ (until $\Pi^{k-1} = \Pi^k$).

state	a	b
1	....2....	....3....
2	....2....	....4....
3	....4....	....5....
4	....2....	....6....
5	....5....	....4....
6	....6....	....6....

Table 3.1. Transition table with the same finite automaton as in Figure 3.1 (automaton M_1).

We present an example using a tabular method to minimize the automaton M_1 (Figure 3.1). We show the partition of states into equivalence classes by drawing horizontal lines.

Example 3.1 Consider the automaton M_1 from Figure 3.1. This automaton is also displayed in the transition table (Table 3.1). Let $[p]_k$ denote the equivalence class of the state p with respect to $\equiv^k$. We also write $[p]$ if $\equiv^k$ is understood.

$\equiv^0$	a	b
B_1: 1	....[2]....	....[3]....
3	....[4]....	....[5]....
4	....[2]....	....[6]....
B_2: 2	....[2]....	....[4]....
5	....[5]....	....[4]....
6	....[6]....	....[6]....

Table 3.2. First partition.

Immediately. from the definition of $\equiv^0$ we have $[1]_0 = [3]_0 = [4]_0 = \{1.3.4\} = B_1$ and $[2]_0 = [5]_0 = [6]_0 = \{2.5.6\} = B_2$. We display this information in the modified transition table (Table 3.2.). in which equivalence classes are separated by horizontal lines. The reason is that when computing $\equiv^1$. we make use of the fact that $p \equiv^1 q$ if and only if $p \equiv^0 q$. $pa \equiv^0 qa$. and $pb \equiv^0 qb$. But this holds if and only if $[p]_0 = [q]_0$. $[pa]_0 = [qa]_0$. and $[pb]_0 = [qb]_0$. In other words. if and only if the rows of p and q are "equivalent".

Consider $[1]_0 = \{1.3.4\}$ first. By inspection. every pair of rows from this set are distinct. More precisely $[1a]_0 \neq [3a]_0$. $[1b]_0 \neq [4b]_0$. and $[3a]_0 \neq [4a]_0$. This implies 1. 3. and 4 form singleton equivalence classes in $\equiv^1$.

On the other hand. rows 2 and 5 are equivalent and different from row 6. Thus. we obtain Table 3.3. To construct $\equiv^2$. we only need to examine the rows 2 and 5. since all other equivalence classes are singletons. Once more these two rows are

$\equiv^1$	a	b
1	....[2]....	....[3]....
3	....[4]....	....[5]....
4	....[2]....	....[6]....
2	....[2]....	....[4]....
5	....[5]....	....[4]....
6	....[6]....	....[6]....

Table 3.3. Second partition.

equivalent. Hence $\equiv^2 = \equiv^1 = \equiv$. and we are done. The final minimal automaton is shown in Figure 3.2.

CHAPTER 4

ALGORITHM DUE TO HOPCROFT AND ULLMAN

state	a	b	c
p	....r....	....t....	....v....
q	....s....	....u...	....w....

Table 4.1. State Table. A pair of states which should be compared at some point in the algorithm

The problem with the algorithm presented in the previous chapter is that it can perform useless comparisons between pairs of states. One can save much work by bookkeeping. Note that, whenever $r \not\equiv s$ for some $r, s \in Q$, then $pw \not\equiv qw$ for every $p, q \in Q$ and $w \in \Sigma^*$, such that $pw = r$ and $qw = s$.

Suppose that $p, q, r, s, t, u, v, w \in Q$ and $\{a, b, c\} \subseteq \Sigma$ and we have found the information shown in Table 4.1 which we will call ***state table***. Assume we know nothing about the pairs $\{(r, s), (t, u), (v, w)\}$ (we don't know if they are distinguishable or not). Then (p, q) are distinguishable or indistinguishable depending on what we find out later about these pairs. So, for every one of these pairs, we place (p, q) on a list linked to the pair. It is called the ***list to mark***. Then, if we later find that one of these pairs is distinguishable, we go through its list and mark pairs on it as distinguishable: in this way, we do not have to check p and q for distinguishability

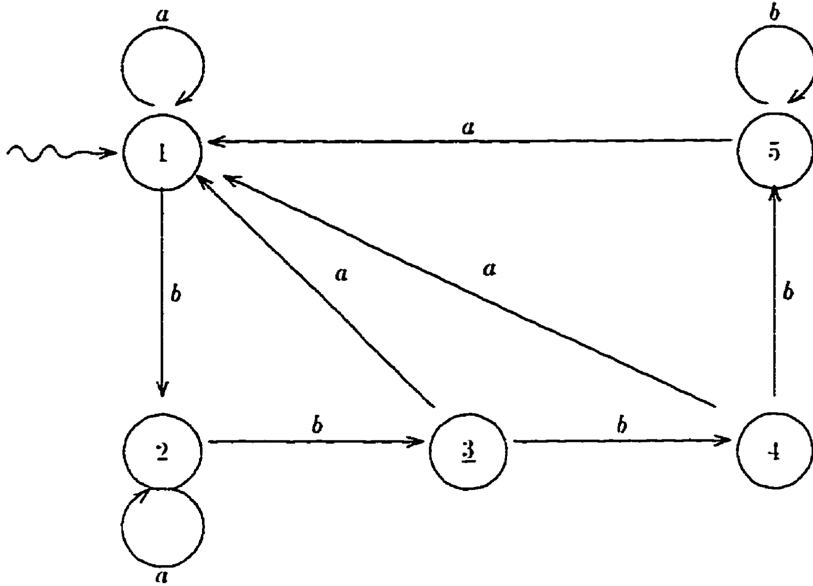

Figure 4.1. Deterministic finite automaton M_2 with the partition of states into two blocks $B_1 = \{1, 2, 4, 5\}$ and $B_2 = \{\underline{3}\}$.

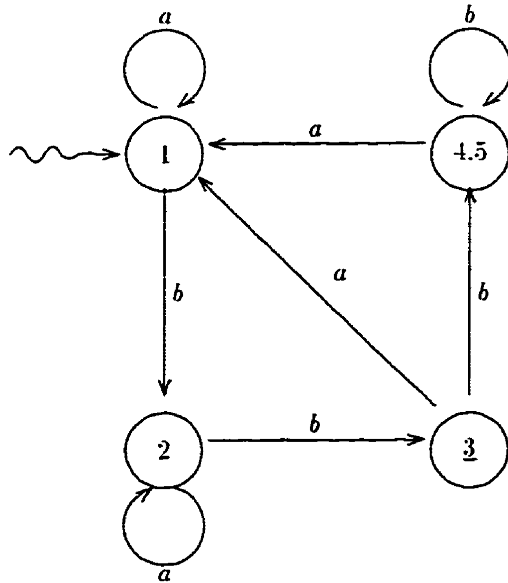

Figure 4.2. Minimum state finite automaton M_2'. This automaton is equivalent to the automaton M_2.

repeatedly. Thus note that, when (p, q) become distinguishable as a consequence of

the distinguishability of (r, s), say, then also the list originating at (p, q) needs to be dealt with. Note also, that a pair can appear on several lists.

One way to implement this algorithm, according to [4], is to use a triangular $(n-1) \times (n-1)$ table D, where n is the number of states. We call this the *distinguishability matrix*. Assume Q is totally ordered in an arbitrary, but fixed way. Thus $Q = \{q_0, q_1, \ldots, q_{n-1}\}$. The entries of this matrix are initialized to zero: and we set the d_{ij} entry of D to one, whenever we establish that q_i and q_j are distinguishable, where $i \leq j$.

To use this method, we must start with one or more pairs of states known to be distinguishable. Clearly, every state $q_i \in B_k$ is distinguishable from the states belonging to different blocks of B, thus it is distinguishable from every $q_j \notin B_k$. This corresponds to the construction of $\equiv^0$ in Wood's algorithm. The states from the same block B_k may or may not be distinguishable from one another: we do not know that yet. Hence in the distinguishability matrix D we go through all rows and mark every column corresponding to a state belonging to a different block. For example if some row i represents a state $q_i \in B_k$, we mark every entry in this row corresponding to a column j representing state $q_j \notin B_k$.

Next, we consider every pair (q_i, q_j) which corresponds to an unmarked entry, $d_{i,j} = 0$. We look at the next-state functions of q_i and q_j (state tables) to see whether there is any input that takes them to a pair known to be distinguishable. If they do, than q_i and q_j are also distinguishable. That is, we compare their rows in the state table. We have three possible outcomes:

a) the rows are identical or equivalent

b) the rows differ by states known to be distinguishable

c) the rows differ by states about which we do not know yet

In case a, we leave their table entry unmarked ($d_{ij} = 0$): in case b we can mark them as distinguishable ($d_{ij} = 1$) and we recursively mark every pair on the list to

mark. in case c we have the situation shown in Table 4.1. and we do what we have proposed at the beginning to avoid additional (useless) comparisons. That means for every input symbol $a \in \Sigma$ we put (q_i, q_j) on the list to mark of $(q_i a, q_j a)$.

When we have checked all unmarked entries. the algorithm terminates. We can identify the sets of equivalent states by examining the entries for each row or column of the matrix. Any zero in the row (column) for state q marks another state that is equivalent to q. Since state equivalence is a true equivalence relation. we can accumulate classes of states by taking advantage of transitivity. The equivalence classes are the states of the new machine.

Below one can see the algorithm:

Algorithm 4.1

Input: A complete DFA with set of states Q. all states reachable. set of input symbols Σ. transitions defined for all states and inputs. start state s. and the partition of states B.

Output: A minimal DFA M' equivalent to M

Method:

1. **For** every $q_i, q_j \in \Sigma$. where $i \leq j$. if $q_i \in B_k$ and $q_j \notin B_k$ **do** mark (q_i, q_j).
2. **For** each unmarked pair (q_i, q_j) **do**
3. **if** for some input symbol a. $(q_i a, q_j a)$ is marked **then**

   begin

   mark (q_i, q_j):

   recursively mark all unmarked pairs on the list to mark for (q_i, q_j) and on the lists to mark of other pairs that are marked at this step.

   end
4. **else**

   for all input symbols a **do**

put (q_i, q_j) on the list to mark for $(q_i a, q_j a)$ unless $q_i a = q_j a$.

5. Construct the minimal automaton using the information from the matrix.

This algorithm is more efficient than the obvious marking algorithm. The time complexity is $O(n^2)$, where n is the number of states.

Example 4.1

	....2....	....3....	....4....	....5....
1	....0....	....1....	....0....	....0....
2	....	....1....	....0....	....0....
3	....	....	....1....	....1....
4	....	....	....	....0....

Table 4.2. Initial distinguishability matrix.

	....2....	....3....	....4....	....5....
1	....1....	....1....	....0....	....0....
2	....	....1....	....0....	....0....
3	....	....	....1....	....1....
4	....	....	....	....0....

Table 4.3. Second distinguishability matrix.

Consider the machine in Figure 4.1. The initial distinguishability matrix is shown in Table 4.2. Here state 3 is the only state belonging to B_2 and as such it is

	2	3	4	5
1	1	1	1	1
2		1	1	1
3			1	1
4				0

Table 4.4. Third distinguishability matrix.

distinguishable from all other states: hence there are ones in the row and column for state 3 and zeros everywhere else. Initially we assume all the other states are equivalent: now we must see whether they really are. We start by considering zero entries for state 1: we must first determine whether it is distinguishable from states 2. 4. and 5.

$\underline{d_{12}}$: Comparing states 1 and 2. we have

state	a	b
1	1	2
2	2	3

We can say that an input b will take the pair (1.2) to the pair (2.3). Since 2 and 3 are known to be distinguishable. states 1 and 2 are distinguishable. too. So $d_{12} = 1$ and our next matrix will be the matrix from Table 4.3.

$\underline{d_{14}}$: When we compare states 1 and 4. we get inconclusive results:

state	a	b
1	1	2
4	1	5

In this case. states 1 and 4 will be distinguishable only if states 2 and 5 are: but we don't know yet. We could come back later. after we have checked 2 and 5. and try again. but for large machines this would be prohibitively time consuming. So

we enter the pair (1.4) in the list to mark associated with (2.5). We can represent this list by writing

(2.5) → (1.4)

If (2.5) later turns out to be distinguishable, then we merely have to go down the list of pairs associated with (2.5) and mark them as distinguishable. If it should happen that 2 and 5 never turn out to be distinguishable, then 1 and 4 won't be, either.

d_{15}: The same happens when we compare States 1 and 5:

state	....a....	....b....
1	....1....	....2....
5	....1....	....5....

So we add (1.5) to the list to mark associated with (2.5). This list is now

(2.5) → (1.4) → (1.5)

d_{24}: Next we consider states 2 and 4 and we have

state	....a....	....b....
2	....2....	....3....
4	....1....	....5....

In this case, a takes (2.4) to the distinguishable pair (2.1), so 2 and 4 are distinguishable. Thus $d_{24} = 1$.

d_{25}: For states 2 and 5, we have

state	....a....	....b....
2	....2....	....3....
5	....1....	....5....

and we see that 2 and 5 are distinguishable. But we have a list attached to (2.5): going down this list, we see that we can mark (1.4) and (1.5) as distinguishable. Thus we set $d_{25} = d_{14} = d_{15} = 1$ and in Table 4.4 one sees the next distinguishability matrix.

d_{45}: Since we know that state 3 is distinguishable from all the other states, we move

on to state 4: the only pair we need to test is (4, 5):

state	....a....	....b....
4	....1....	....5....
5	....1....	....5....

Since the two rows are identical, we cannot distinguish states 4 and 5 and hence conclude that they are equivalent. So our final matrix is still Table 4.4. Our minimal-state machine is, therefore, illustrated in Figure 4.2.

In programming this procedure, the easiest way to handle the chaining of undecided states is to make the distinguishability matrix a matrix of records, with a field for the mark (1 or 0) and a field for a pointer to the chain. The matrix entries are initially set to 0 and *nil*. After marking all nonaccepting states as distinct from all accepting states by putting 1 in the appropriate entries of the matrix, one goes through the matrix. If a pair is not immediately distinguishable, it is added to the lists to mark of all next-state pairs, except where the two next states are identical. If a pair is found to be distinguishable, one marks the matrix entry with a 1 and then follows up the chain, if any. Notice that the chains may branch, so the follow-up procedure must be made recursive. This recursive procedure always terminates, as there is a finite number of the pairs of states which can be marked. If a pair is already marked its *list to mark* is not considered.

CHAPTER 5

NEW ALGORITHMS

In this chapter three new algorithms for minimization are presented. They are called: General Version. Second Version and Third Version. All these versions of the minimization algorithm are based on the same concept. These algorithms assign names to all states in an automaton. Then they sort the names of states of an automaton to minimize it. After sorting. if some states are renamed. the list is sorted again. The names of states have the following property: if at step k of the algorithm states have distinct names. they have distinct names at every step $k+i$. where $i \geq 1$. The algorithms terminate if. as a result of sorting. the list is not changed (states are not renamed).

The General Version of the minimization algorithm sorts the whole list of names of states at every step of the algorithm. The upper bound for this version is $O(n^2 \log n)$. The General Version of the minimization algorithm is not an efficient one. It does not use the information that. as a result of sorting at step k. at step $k+1$ we already know some pairs of states which should have distinct names. Thus there is no need to check if they are still distinct by sorting the whole list. The Second Version of the algorithm uses this information and it sorts the whole list of names class by class as one class of states has different names from all others classes of states). The time complexity for this version of the minimization algorithm is

$\Theta(n^2)$.

The Second Version of the minimization algorithm may still sort a class of states which have the same name. Thus no changes are made. In the Third Version of the algorithm only initially all classes are sorted. At the next steps of this algorithm only classes of states with distinct names are sorted. This version has the same time complexity as the Second Version of the minimization algorithm. $\Theta(n^2)$.

5.1. General Version

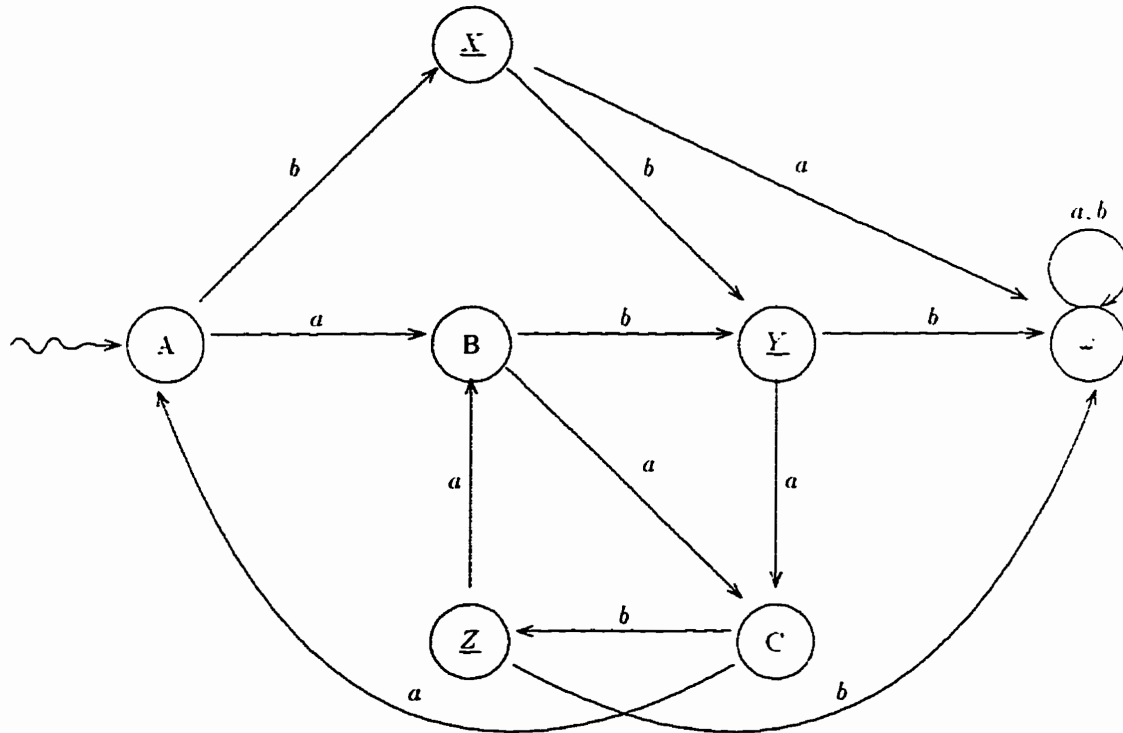

Figure 5.1.1. Deterministic finite automaton M_1' with $B = (\omega, \{A, B, C\}, \{\underline{X}, \underline{Y}, \underline{Z}\})$.

The new algorithm for minimization is based on Wood's algorithm. Wood's algorithm works as follows: One starts with $\equiv^0 = \varrho_B$, as two states are 0-indistinguishable if and only if they are in the same block of B. There are $r = |B|$ equivalence classes

formed by $\equiv^0$. We construct $\equiv$ by refining $\equiv^0$ to give $\equiv^1$, $\equiv^1$ to give $\equiv^2$ and so on. This continues until $\equiv^k = \equiv^{k+1}$.

The main idea used in the new version of the minimization algorithm is to give unique names to the equivalence classes formed at every step of the algorithm. At step k of the algorithm every state p belonging to the same equivalence class will also have the same name, and states belonging to different classes will have different names. By N_p^k we denote the name of the equivalence class to which the state p belongs at step k of the algorithm. Let $N^k = \{N_{(1)}^k, N_{(2)}^k, \ldots, N_{(u)}^k\}$ be the set of classes of states at step k of the algorithm. This set is in sorted order and that is why we have subscripts $1, 2, \ldots, u$ where $u \leq |Q|$. Whenever p and q are in the same class $N_{(i)}^k$, they have the same name. We use the symbol $N_{(i)}^k$ to denote the name of the ith class of N^k (the class $N_{(i)}^k$). Thus, if $p \equiv^{k-1} q$, then $p, q \in N_{(i)}^k$ for some i and $N_p^k = N_q^k = N_{(i)}^k$, and if $p \not\equiv^{k-1} q$ then $N_p^k \neq N_q^k$. Whenever an equivalence class is changed (class is divided), the states from this class will get new names. $|N^k|$ represents the number of distinct names of states (number of equivalence classes) at step k of the algorithm.

Suppose we are given an automaton $M = (Q, \Sigma, \delta, s, B)$. Let $\Sigma = (a_1, a_2, \ldots, a_m)$, where the input symbols are ordered in an arbitrary but fixed way. With every state $p \in Q$, we associate the *pattern* P_p^k at step k.

$$P_p^k = \left(N_p^k, N_{pa_1}^k, N_{pa_2}^k, \ldots, N_{pa_m}^k\right)$$

Let $\mathcal{P}^k = \{\mathcal{P}_{(1)}^k, \mathcal{P}_{(2)}^k, \ldots, \mathcal{P}_{(u)}^k\}$, where $u \leq |Q|$, denote the partition of states at step k of the algorithm according to their patterns. $|\mathcal{P}^k|$ is the number of distinct patterns at step k of the algorithm and note that $|N^k| \leq |\mathcal{P}^k|$ for every $k \geq 0$. Thus N^k corresponds to $\equiv^{k-1}$ and $\mathcal{P}^k$ corresponds to $\equiv^k$.

To avoid the calculation of patterns before every sorting, states can be represented by *state structures*: the state structure of a state p consists of the current

name N_p^k of p. the new name N_p^{k+1}. and the transition list of p. The transition list of every state is implemented by pointers such that the pointer for pa_i in the state structure of p points to the state structure of the state pa_i. Thus a list of state structures contains the information about the names of states and the patterns. The current names of states are used to calculate patterns. The new names of states are calculated at step k of the algorithm but they are used in the construction of patterns at $k+1$ step of the algorithm.

After sorting. the list of state structures is ordered in some predefined way. We will call this sorted list *StateStructures*. Now the algorithm can be presented. We assume that all states are reachable.

Algorithm 5.1.1

Input: A complete DFA with set of states Q. all states reachable. set of input symbols Σ. transitions defined for all states and inputs. start state s. and the partition of states B.

Output: A minimal DFA M' equivalent to the DFA M.

Method:

1. Using the function GetNewName associate the name N_{B_i} with every block B_i. So if $p \in B_i$ then $N_p^1 = N_{B_i}$. Let N_{last} be the last name calculated by GetNewName.
2. Sort patterns.
3. WasRenamed := False. /* initially no states were renamed */
4. RenameAll(StateStructures.WasRenamed.N_{last}).
5. **If** any state was renamed (WasRenamed = True) **then** repeat from step (2) **else** terminate the algorithm and construct the minimal automaton by partitioning the states according to their names.

Function GetNewName(N_{last})

/* Given the name N this function calculates and returns a new name N_{new}. If N and all other names currently used in the main program were calculated using this function. all these names including N_{new} are distinct. It is assumed that N_{last} is the last name calculated previously using this function. One possible function f which can be used is (assuming that N is an integer): $f(N) = N + 1$. */

begin

$N_{\text{new}} := f(N_{\text{last}})$

GetNewName := N_{new}

end

Procedure RenameAll(StateStructures. WasRenamed. N_{last})

/* Given the sorted list of patterns. encoded in the list of state structures. this procedure renames appropriate states in such a way that whenever states have different patterns they have different names. and whenever they have the same patterns they have the same names. It is assumed that the last name calculated by GetNewName was N_{last}. WasRenamed is set to true if GetNewName is called at least once. */

begin

$N := N^k_{p_1}$

$N^{k+1}_{p_1} := N$

for every $i \in \{2.3.\ldots.|Q|\}$ **do**

begin

if $P^k_{p_{i-1}} \neq P^k_{p_i}$ and $N^k_{p_{i-1}} \neq N^k_{p_i}$ **then**

$N := N^k_{p_i}$

else if $P^k_{p_{i-1}} \neq P^k_{p_i}$ and $N^k_{p_{i-1}} := N^k_{p_i}$ **then**

begin

WasRenamed := True

N := GetNewName(N_{last})

$N_{\text{last}} := N$

end

$N_{p_i}^{k+1} := N$

end

end

The function GetNewName returns a "new" name (that means a name which is not currently used by any state). To accomplish that we use the function GetNewName every time a new name has to be introduced, and we pass to this function the "last" name (N_{last}) it has calculated. The name N_{last} can be initially set to 0 and GetNewName can use consecutive binary numbers for names. Thus, if we use GetNewName at step 1 of the algorithm, and as a result we have three blocks of states named 0, 1, and 01, GetNewName(01) will return 10 (01 + 1 = 10). As the maximum number of distinct names required in this algorithm is n, the longest binary number used to represent the names has the length at most $\log n$.

Initially we have $|N^1|$ classes, where $|N^1| = |B|$. At every step k of the algorithm new equivalence classes are formed and compared to the classes formed at step $k-1$. Renaming is done in the following way. Suppose that at step $k-1$ of the algorithm, states p and q are in the same class and thus they have the same name. If at the next step the states p and q have different patterns, they should be distinguished and so at least one state: p or q should get a new name at this step. If at step k all states from some class have the same pattern, there is no need to change their names. The procedure RenameAll accomplishes that. Whenever the procedure GetNewName is called at least once, that means some states were renamed and in the algorithm we repeat steps (2). Notice that the list of states structures passed to this procedure

is sorted by patterns. As patterns are constructed from the names of states and the first element in the given pattern is the name of the state. the names of states are also sorted. Notice that in the procedure RenameAll by renaming states we are changing the patterns. But the information about old patterns should be available until the end of this procedure. One way to handle that is to keep two names in the state structure. The first one should represent N_p^k and should be used every time the pattern is calculated. The second one should represent a new name N_p^{k+1}.

Next we see an example. Then we will prove that this algorithm always terminates and that the automaton constructed using this algorithm is minimal.

N^1 ($\equiv^0$)	$p(N_p^1)$	$pa(N_{pa}^1)$	$pb(N_{pb}^1)$
$N_{(1)}^1 = 1$:	$\omega(1)$	$\omega(1)$	$\omega(1)$
$N_{(2)}^1 = 2$:	$A(2)$	$B(2)$	$\underline{X}(3)$
	$C(2)$	$A(2)$	$\underline{Z}(3)$
	$B(2)$	$C(2)$	$\underline{Y}(3)$
$N_{(3)}^1 = 3$:	$\underline{X}(3)$	$\omega(1)$	$\underline{Y}(3)$
	$\underline{Y}(3)$	$C(2)$	$\omega(1)$
	$\underline{Z}(3)$	$B(2)$	$\omega(1)$

Table 5.1.1.

Example 5.1.1 We illustrate the algorithm applied to the automaton M_1 in Figure 5.1.1. This is a minimal automaton and the example shows that the algorithm can verify this fact. In this example we are not following directly the procedure RenameAll. When the block of states is divided into k blocks. procedure RenameAll renames only $k-1$ blocks not changing the name of one block. In this example all such blocks will be renamed.

N^2 ($\equiv^1$)	$p(N_p^2)$	$pa(N_{pa}^2)$	$pb(N_{pb}^2)$
$N_{(1)}^2 = 1$:	$\omega(1)$	$\omega(1)$	$\omega(1)$
$N_{(2)}^2 = 2$:	$A(2)$	$B(2)$	$\underline{X}(31)$
	$C(2)$	$A(2)$	$\underline{Z}(32)$
	$B(2)$	$C(2)$	$\underline{Y}(32)$
$N_{(3)}^2 = 31$:	$\underline{X}(31)$	$\omega(1)$	$\underline{Y}(32)$
$N_{(4)}^2 = 32$:	$\underline{Y}(32)$	$C(2)$	$\omega(1)$
	$\underline{Z}(32)$	$B(2)$	$\omega(1)$

Table 5.1.2.

N^3 ($\equiv^2$)	$p(N_p^3)$	$pa(N_{pa}^3)$	$pb(N_{pb}^3)$
$N_{(1)}^3 = 1$:	$\omega(1)$	$\omega(1)$	$\omega(1)$
$N_{(2)}^3 = 21$:	$A(21)$	$B(22)$	$\underline{X}(31)$
$N_{(3)}^3 = 22$:	$C(22)$	$A(21)$	$\underline{Z}(32)$
	$B(22)$	$C(22)$	$\underline{Y}(32)$
$N_{(4)}^3 = 31$:	$\underline{X}(31)$	$\omega(1)$	$\underline{Y}(32)$
$N_{(5)}^3 = 32$:	$\underline{Y}(32)$	$C(22)$	$\omega(1)$
	$\underline{Z}(32)$	$B(22)$	$\omega(1)$

Table 5.1.3.

Thus we are given a DFA M_1 with the partition of states of this automaton $B = (B_1, B_2, B_3) = (\{\omega\}, \{A, B, C\}, \{\underline{X}, \underline{Y}, \underline{Z}\})$. Since B is a partition, each state p is in exactly one block B_j. We assign the initial name N_p^1 to each state $p \in Q$ in

N^4 ($\equiv^3$)	$p(N_p^4)$	$pa(N_{pa}^4)$	$pb(N_{pb}^4)$
$N_{(1)}^4 = 1$:	$\omega(1)$	$\omega(1)$	$\omega(1)$
$N_{(2)}^4 = 21$:	$A(21)$	$B(22)$	$\underline{X}(31)$
$N_{(3)}^4 = 221$:	$C(221)$	$A(21)$	$\underline{Z}(32)$
$N_{(4)}^4 = 222$:	$B(222)$	$C(221)$	$\underline{Y}(32)$
$N_{(5)}^4 = 31$:	$\underline{X}(31)$	$\omega(1)$	$\underline{Y}(32)$
$N_{(6)}^4 = 32$:	$\underline{Y}(32)$	$C(221)$	$\omega(1)$
	$\underline{Z}(32)$	$B(222)$	$\omega(1)$

Table 5.1.4.

N^5 ($\equiv^4$)	$p(N_p^5)$	$pa(N_{pa}^5)$	$pb(N_{pb}^5)$
$N_{(1)}^5 = 1$:	$\omega(1)$	$\omega(1)$	$\omega(1)$
$N_{(2)}^5 = 21$:	$A(21)$	$B(22)$	$\underline{X}(31)$
$N_{(3)}^5 = 221$:	$C(221)$	$A(21)$	$\underline{Z}(322)$
$N_{(4)}^5 = 222$:	$B(222)$	$C(221)$	$\underline{Y}(321)$
$N_{(5)}^5 = 31$:	$\underline{X}(31)$	$\omega(1)$	$\underline{Y}(321)$
$N_{(6)}^5 = 321$:	$\underline{Y}(321)$	$C(221)$	$\omega(1)$
$N_{(7)}^5 = 322$:	$\underline{Z}(322)$	$B(222)$	$\omega(1)$

Table 5.1.5.

the following way:

$$N_p^1 = j \text{ if } p \in B_j$$

This gives the following initial names to every state:

$$N_{\omega}^{1} = 1$$

$$N_{A}^{1} = N_{B}^{1} = N_{C}^{1} = 2$$

$$N_{\underline{X}}^{1} = N_{\underline{Y}}^{1} = N_{\underline{Z}}^{1} = 3$$

We use the tabular method to compute $\equiv^{k}$ from $\equiv^{k-1}$ using sorting. We illustrate the state structure using the following format:

$$N_{p}^{k} \quad pa(N_{pa}^{k}) \quad pb(N_{pb}^{k})$$

where p. pa. and pb denote pointers . and N_{p}^{k}. N_{pa}^{k}. and N_{pb}^{k} denote the current names of the states p. pa. and pb. Remember that the numbers before brackets (pointers) and the brackets are not needed for sorting purposes. Also notice that the comparisons are made between elements N_{p}^{k} which are included in the pattern. and not between other elements. So every N_{p}^{k} in the pattern is treated as one element. not as a sequence of symbols or letters. It should be remembered that individual N_{p}^{k} have different lengths because of the renaming.

The initial situation is shown in Table 5.1.1. After sorting the names of the states. we have only 3 equivalence classes. But when we sort the patterns. we see that $\{\underline{X}\}$ and $\{\underline{Y}.\underline{Z}\}$ should be distinguished. So the class 3 should be divided into two classes. and the names of the states should be changed to 31 and 32 to get $\equiv^{1}$. We write a shorter horizontal line to denote that at the next step of the algorithm. the class should be divided. and so states in this class should be renamed. The results of the next sorting steps are shown in Table 5.1.2. Table 5.1.3. Table 5.1.4. and Table 5.1.5. Finally we see that $\equiv^{4}=\equiv^{5}$ ($|N^{5}| = |\mathcal{P}^{5}|$) and we cannot combine any states (no two states are equivalent) and so the automaton M_1 is minimal.

Theorem 5.1.1 *Let $M = (Q, \Sigma, \delta, s, B)$. Algorithm 5.1.1 terminates in at most $n-2$ steps, where $n = |Q|$.*

Proof: The algorithm terminates when $\equiv^k = \equiv^{k+1}$, for some $k \geq 1$. First we have to prove that whenever $\equiv^k = \equiv^{k+1}$, then $\equiv^k = \equiv^{k+i}$, for all $i \geq 0$. If $\equiv^k = \equiv^{k+1}$, then $p \equiv^k q$ implies that $P_p^k = P_p^k$ and so $pa \equiv^k qa$ for every $a \in \Sigma$. But this implies that $p \equiv^{k+2} q$. By induction $\equiv^k = \equiv^{k+i}$, for all $i \geq 0$.
If $|N^1| = 1$, then the algorithm terminates with $\equiv = \equiv^0$, having one equivalence class containing all states. In this case all states are indistinguishable. Otherwise $|N^1| \geq 2$, and $2 \leq |N^k| \leq n$. At every step of the algorithm we have at least one additional class. Thus the main loop in the algorithm is called at most $n-2$ times.

Corollary 5.1.1 *Let $M = (Q, \Sigma, \delta, s, B)$ be a complete DFA having only reachable states and $M' = (Q', \Sigma, \delta', s', B')$ be the corresponding DFA constructed using Algorithm 5.1.1. Then M' is a minimal complete DFA equivalent to M.*

Theorem 5.1.2 *The runtime of Algorithm 5.1.1 is bounded from above by $O(n^2 \log n)$.*

Proof: The algorithm terminates in at most $n-2$ steps where $n = |Q|$. At every step one sorting procedure is performed. It requires at most $n \log n$ comparisons. This gives the required upper bound $O(n^2 \log n)$.

This algorithm is based on sorting of states. At every step of the algorithm names of states are sorted once. Thus the minimal time for the algorithm can not be less than the time required to sort the list of patterns once (to check if $\equiv^0 = \equiv^1$). It will be $O(n \log n)$ if and only if all patterns are distinct. Otherwise it depends on the number of distinct patterns. Assume that $|N^1| \neq 1$ as the case of $|N^1|$ is trivial. Consider the example when $B = \{B_1, B_2\}$ and the block B_1 contains only one state. Suppose that in the minimal automaton we have just 2 states. This corresponds to

having n elements to sort, where only one is different from all others. This implies that patterns are sorted just once. Using mergesort when we merge any 2 lists we can combine the "same" elements, so the maximum number of elements in the lists to be merged is 2 and the maximum number of comparisons is $n + \log n$. This implies that the best case for this algorithm has time complexity $O(n)$.

It is difficult to describe precisely which automaton will represent the worst case as, the more comparisons are made at every step of the algorithm (during sorting), the fewer potential steps can be done in this algorithm. Also, during consecutive steps of the algorithm the list of states is more "in order", so fewer comparisons need to be used to sort it. Consider the example when, using mergesort, $n \log n$ comparisons are made at the first step of the algorithm. That implies that all states are distinguishable and so the main loop of the algorithm can be executed at most once more.

5.2. Second Version

Wood's algorithm does not check every pair of states at every step of the algorithm. Only the pairs of states which are indistinguishable at the given step indistinguishable are checked to decide if they can be distinguished. Similarly Algorithm 5.1.1 can be improved by requiring that, at every step k of the algorithm, states having the same names are sorted separately (these are the classes of $(k-1)$-indistinguishable states). Thus patterns of states belonging to the same block $\mathcal{N}^k_{(i)}$ are sorted separately. By *StateStructures*($\mathcal{N}^k_{(i)}$) denote the set of state structures of the states belonging to the same block $\mathcal{N}^k_{(i)}$ of $\mathcal{N}^k$ at step k of the algorithm.

Algorithm 5.2.1

Input: A complete DFA with set of states Q, all states reachable, set of input symbols Σ, transitions defined for all states and inputs, start state s, and the partition of states B.

Output: A minimal DFA M' equivalent to the DFA M.

Method:

1. Using the function GetNewName associate the name N_{B_i} with every block B_i. So if $p \in B_i$ then $N_p^1 = N_{B_i}$. Let N_{last} be the last name calculated by GetNewName.
2. WasRenamed := False. /* initially no states are renamed */
3. **For** every block of states $N_{(i)}^k$ **do**

   begin

   sort patterns

   RenameBlock(StateStructures($N_{(i)}^k$).WasRenamed.N_{last})

   end.
4. **If** any state was renamed (if WasRenamed = True) **then** repeat from step (2) **else** terminate the algorithm and construct the minimal automaton by partitioning the states according to their names.

Procedure RenameBlock(StateStructures($N_{(i)}^k$).WasRenamed.N_{last})

/* This procedure is given the sorted list of patterns of states encoded in the list of state structures. These states have the same name $N_{(i)}^k$. The procedure renames appropriate states in such a way that whenever states have different patterns they get different names. and whenever they have the same patterns they get the same names. It is assumed that the last name returned by GetNewName is N_{last}. WasRenamed is set to true when the function GetNewName is called at least once. */

begin

$N := N_{p_1}^k$

$N_{p_1}^{k+1} := N$

for every $i \in \{2.3.\dots.|N_{(i)}^k|\}$ **do**

begin

if $P^k_{p_{i-1}} \neq P^k_{p_i}$ **then**

begin

WasRenamed := True

N :=GetNewName(N_{last})

$N_{\text{last}} := N$

end

$N^{k+1}_{p_i} := N$

end

end

Renaming is done in a way similar to the general version of the algorithm. The procedure RenameBlock is different from RenameAll only because, to the procedure RenameBlock, the information about state structures of only one block $N^k_{(i)}$ is passed (not the information about state structures of all states). Thus, initially, the names of the states are the same but, because patterns can differ, the procedure RenameBlock can assign different names to some states.

Theorem 5.2.1 *The runtime of Algorithm 5.2.1 is* $\Theta(n^2)$.

Proof: The length of the lists of patterns which are merged during mergesort can not be more than the number of distinct classes in the given set of patterns which has to be sorted. If we assume that the number of distinct patterns is g, the lists to be merged can not be longer than g. So the number of comparisons required to merge two lists is no more than $2g - 1$. As the lists are merged at most n times during mergesort, the total number of comparisons required by mergesort is bounded by $(2g - 1)n$. The number of comparisons is the highest when at each step of the algorithm only one additional class is formed. So after sorting any class we

have at most $g = 2$ distinct classes. As in this version of the algorithm we sort class by class, at each step of the algorithm we have at least 1 merge less to do. So the maximum number of comparisons is:

$$(2g-1)n + (2g-1)(n-1) + \ \dots\ + (2g-1)2 + (2g-1)$$

where $g = 2$. Thus the upper bound is $O(n^2)$.

To show that $\Theta(n^2)$ is the time complexity for this algorithm, consider the automaton $M_{\text{worst}} = (Q, \Sigma, \delta, s, B)$ where $Q = \{p_1, p_2, \dots, p_n\}$, $s = p_1$, $\Sigma = \{a, b\}$, $B = \{\{p_1, p_2, \dots, p_{n-1}\}, \{p_n\}\}$, and

$$\delta(p_i, a) = \begin{cases} p_{i-1} & \text{for } i \geq 2 \\ p_1 & \text{for } i = 1 \end{cases}$$

$$\delta(p_i, b) = p_n$$

An example of such automaton for $n = 6$ is shown in Figure 5.2.1. This represents the case when $B = \{B_1, B_2\}$, and only $\underline{Z}$ is in B_2. To minimize this automaton, or rather to use the algorithm to verify that the automaton is minimal, the patterns have to be sorted $n-2$ times (compare Table 5.2.1 and Table 5.2.2). Considering the automaton M_{worst}, at step k of the algorithm we have k singleton classes and one class $X^k_{(i)}$ containing $n-k$ elements. It is possible to eliminate the sorting of the singleton classes, but all elements in the class with $n-k$ elements should be checked at the given step of the algorithm. Thus we need at least $n-1$, $n-2$, $n-3$, $n-4, \dots$ comparisons. This means that the algorithm has time complexity $\Omega(n^2)$ and so it has time complexity $\Theta(n^2)$.

The best case is the same as in the general version of the algorithm. This is the case when in the minimal automaton there are only two states. This corresponds to having n elements to sort, where only one is different from all others. Thus patterns are sorted just once and the maximum number of comparisons is $n + \log n$. This

implies that the best case for this algorithm has time complexity $O(n)$ similarly to the general version of the minimization algorithm.

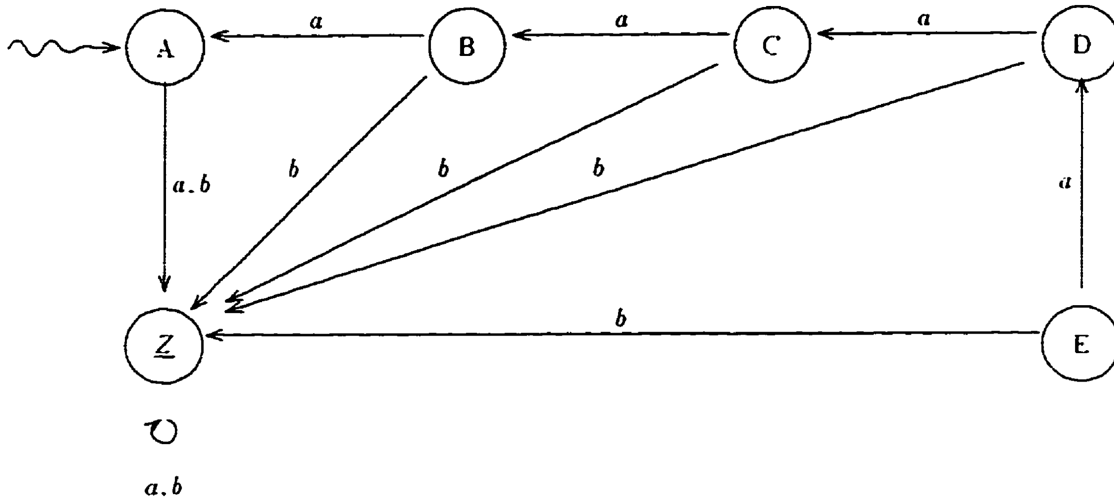

Figure 5.2.1. Deterministic finite automaton M_4 for which it takes $\Omega(n^2)$ comparisons to verify that it is minimal using the second or the third versions of the algorithm. The states in this automaton are partitioned into two blocks $B_1 = \{A, B, C, D, E\}$ and $B_2 = \{\underline{Z}\}$.

$X^1\ (\equiv^0)$	$p(X^1_p)$	$pa(X^1_{pa})$	$pb(X^1_{pb})$
$X^1_{(1)} = 1$:	$E(1)$	$D(1)$	$\underline{Z}(2)$
	$D(1)$	$C(1)$	$\underline{Z}(2)$
	$C(1)$	$B(1)$	$\underline{Z}(2)$
	$B(1)$	$A(1)$	$\underline{Z}(2)$
	$A(1)$	$\underline{Z}(2)$	$\underline{Z}(2)$
$X^1_{(2)} = 2$:	$\underline{Z}(2)$	$\underline{Z}(2)$	$\underline{Z}(2)$

Table 5.2.1.

X^2 ($\equiv^1$)	$p(X^2_p)$	$pa(X^2_{pa})$	$pb(X^2_{pb})$
$X^2_{(1)} = 11$:	$E(11)$	$D(11)$	$\underline{Z}(2)$
	$D(11)$	$C(11)$	$\underline{Z}(2)$
	$C(11)$	$B(11)$	$\underline{Z}(2)$
	$B(11)$	$A(12)$	$\underline{Z}(2)$
$X^2_{(2)} = 12$:	$A(12)$	$\underline{Z}(2)$	$\underline{Z}(2)$
$X^2_{(3)} = 2$:	$\underline{Z}(2)$	$\underline{Z}(2)$	$\underline{Z}(2)$

Table 5.2.3.

X^3 ($\equiv^2$)	$p(X^3_p)$	$pa(X^3_{pa})$	$pb(X^3_{pb})$
$X^3_{(1)} = 111$:	$E(111)$	$D(111)$	$\underline{Z}(2)$
	$D(111)$	$C(111)$	$\underline{Z}(2)$
	$C(111)$	$B(111)$	$\underline{Z}(2)$
$X^3_{(2)} = 112$:	$B(112)$	$A(12)$	$\underline{Z}(2)$
$X^3_{(3)} = 12$:	$A(12)$	$\underline{Z}(2)$	$\underline{Z}(2)$
$X^3_{(4)} = 2$:	$\underline{Z}(2)$	$\underline{Z}(2)$	$\underline{Z}(2)$

Table 5.2.4.

5.3. Third Version

In the second version of the algorithm the states are sorted class by class. It can happen that we sort a class in which all states have the same pattern. There is no need to sort such classes. Thus, similarly to the algorithm due to Hopcroft and

X^4 ($\equiv^3$)	$p(X^4_p)$	$pa(X^4_{pa})$	$pb(X^4_{pb})$
$X^4_{(1)} = 1111$:	$E(1111)$	$D(1111)$	$\underline{Z}(2)$
	$D(1111)$	$C(1112)$	$\underline{Z}(2)$
$X^4_{(2)} = 1112$:	$C(1112)$	$B(112)$	$\underline{Z}(2)$
$X^4_{(3)} = 112$:	$B(112)$	$A(12)$	$\underline{Z}(2)$
$X^4_{(4)} = 12$:	$A(12)$	$\underline{Z}(2)$	$\underline{Z}(2)$
$X^4_{(5)} = 2$:	$\underline{Z}(2)$	$\underline{Z}(2)$	$\underline{Z}(2)$

Table 5.2.5.

X^5 ($\equiv^4$)	$p(X^5_p)$	$pa(X^5_{pa})$	$pb(X^5_{pb})$
$X^5_{(1)} = 11111$:	$E(11111)$	$D(11112)$	$\underline{Z}(2)$
	$D(11112)$	$C(1112)$	$\underline{Z}(2)$
$X^5_{(2)} = 1112$:	$C(1112)$	$B(112)$	$\underline{Z}(2)$
$X^5_{(3)} = 112$:	$B(112)$	$A(12)$	$\underline{Z}(2)$
$X^5_{(4)} = 12$:	$A(12)$	$\underline{Z}(2)$	$\underline{Z}(2)$
$X^5_{(5)} = 2$:	$\underline{Z}(2)$	$\underline{Z}(2)$	$\underline{Z}(2)$

Table 5.2.2.

Ullman, we want to deal only with the states which have to be distinguished. Let K and L be the lists of classes of the states to be sorted. Thus at step k of the algorithm $L = \{X^k_{(1)}, X^k_{(2)}, \ldots, X^k_{(t)}\}$ where $t \leq |X^k| \leq |Q|$. In the algorithm, classes from the list L are sorted by patterns taking one class at a time, and at the same

time the new list K is built. Only initially can it happen that we sort classes in which all patterns are the same. Then we sort only classes in which some states have different patterns.

Algorithm 5.3.1

Input: A complete DFA with set of states Q, all states reachable, set of input symbols Σ, transitions defined for all states and inputs, start state s, and the partition of states B.

Output: A minimal DFA M' equivalent to the DFA M.

Method:

1. Using the function CalculateNewName associate the name N_{B_i} with every block B_i. So if $p \in B_i$ then $N_p^1 = N_{B_i}$. Let N_{last} be the last name calculated by CalculateNewName.
2. Let $K := L := \emptyset$. /* initially the lists K and L are empty */
3. Put all classes $N_{(i)}^1$ on the list to sort L.
4. **For** every class of states $N_{(i)}^k$ from the list to sort L **do**

   begin

   sort patterns

   TotallyRenameBlock(StateStructures($N_{(i)}^k$),N_{last}) /* rename states */.

   UpdateListToSortK(StateStructures, ClassStruct($N_{(i)}^k$),K)

   end.
5. Let $L := K$ and $K := null$.
6. **If** $L \neq \emptyset$ **then** repeat from step (4)

   else terminate the algorithm and construct the minimal automaton by partitioning the states according to their names.

In this algorithm the following structures and procedures are used:

Structure StateStruct(q)

/* The structure for a state q holds the following information:

1. The result of applying a single input symbol to the state p. This information is contained in the transition list which is a list of pointers to appropriate state structures. It is assumed that $\Sigma = \{a_1, a_2, \ldots, a_m\}$, where input symbols are ordered in arbitrary but fixed way.
2. The information about every predecessor state p. This information is contained in the connection set which contains the pointers to the appropriate state structures.
3. The pointer to the structure, ClassStruct*($X^{*k}_{(i)}$) which contains information about the class $X^{*k}_{(i)}$ to which the state q belongs at the given step of the algorithm. */

transition list= (StateStruct*(qa_1), StateStruct*(qa_2), ..., StateStruct*(qa_n))

connection set= { (StateStruct*(p), a) $|p \in Q, a \in \Sigma, pa = q$ }

ClassStruct*($X^{*k}_{(i)}$)

Structure ClassStruct($X^{*k}_{(i)}$)

/* The structure for a class $X^{*k}_{(i)}$ holds the following information:

1. Name of the class $X^{*k}_{(i)}$ (used to calculate the pattern at step k).
2. Name of the class $X^{*k+1}_{(j)}$ (used to calculate the pattern at step $k+1$).
3. The information about all states belonging to the class $X^{*k}_{(i)}$. This information is contained in the set of states. The set of states contains the pointers to the state structures of appropriate states.
4. The information about the current names of successor states of the states belonging to the class $X^{*k}_{(i)}$ (If names of successors states are different for some input symbol a_i, that means that patterns are different and so the class $X^{*k}_{(i)}$ should be divided). It is assumed that $\Sigma = \{a_1, a_2, \ldots, a_m\}$ where input

symbols are ordered in arbitrary but fixed way. */

$N^k_{(i)}$ /* name of a class and so name of all states belonging to this class */

$N^{k+1}_{(j)}$ /* new name of this class */

set of states = { StateStruct*(z) | $z \in N^k_{(i)}$ }

marking list = (mark(a_1), mark(a_2), ..., mark(a_n))

Function CalculateNewName(N_{last}, N_{prev})

/* Given the names N_{last} and N_{prev}, this function calculates and returns a new name N_{new}. The name N_{last} is the last name calculated using this function. The name N_{prev} is the current name of the state/class for which the new name is to be calculated. If N and all other names currently used in the main program were calculated using this function, all these names including N_{new} are distinct. The function f should have the inverse function g with the property that if $f(N_{\text{last}}, N_{\text{prev}}) = N_{new}$ then $g(N_{new}) = N_{\text{prev}}$. */

begin

$N_{new} := f(N_{\text{last}}, N_{\text{prev}})$

CalculateNewName := N_{new}

end

Procedure TotallyRenameBlock(StateStructures($N^k_{(i)}$),N_{last})

/* This procedure is given the sorted list of patterns of states encoded in the list of state structures. These states have the same name $N^k_{(i)}$. The procedure renames all states in such a way that whenever states have different patterns they get different names, and whenever they have the same patterns they get the same names. It is assumed that the last name returned by CalculateNewName is N_{last}. */

begin

N := CalculateNewName(N_{last}, $N^k_{p_1}$) /* construct a new ClassStruct */

$N^{k+1}_{p_1} := N$

for every $i \in \{2, 3, \ldots, |X^k_{(i)}|\}$ **do**

begin

if $P^k_{p_{i-1}} \neq P^k_{p_i}$ **then**

begin

X :=CalculateNewName($X_{\text{last}}, X^k_{p_i}$) /* construct a new ClassStruct */

$X_{\text{last}} := X$

end

$X^{k+1}_{p_i} := X$

end

end

Procedure UpdateListToSortK(StateStructures, ClassStruct($X^k_{(i)}$), K)

/* Given the information about the class in which the states have been just renamed this procedure updates the list to sort K. If it detects that patterns of states in some class $X^k_{(j)}$ become distinct (because of the renaming of states from the block $X^k_{(j)}$), it puts this class at the list to sort K. StateStructures is a pointer to the linked list containing information about all state structures and class structures (StateStruct and ClassStruct). ClassStruct($X^k_{(i)}$) is a pointer to a structure containing the information about the class which has been just renamed. */

for every StateStruct(q) from the set of states in ClassStruct($X^k_{(i)}$)

/* Thus $q \in X^k_{(i)}$. Assume $q \in X^{k+1}_{(l)}$ */

begin

for every (StateStruct(p),a) from the connection set in StateStruct(q)

/* Assume $p \in X^r_{(j)}$, where $r = k$ or $r = k+1$ */

if $X^r_{(j)} \notin L$

begin

if in StateStruct($X^r_{(j)}$) mark(a)= X and $X \neq X^{k+1}_{(i)}$ and $g(X) = g(X^{k+1}_{(i)})$

put $N^r_{(j)}$ in the list to sort K

else

mark$(a) := N^{k+1}_{(i)}$

end

end

In Algorithm 5.3.1 initially all patterns are sorted. but then we want to make sure that at the next step we sort only the classes of states with distinct names and so the classes which will be divided. One way to implement this is to introduce two structures: the ***state structure*** and the ***class structure***. They will be denoted by StateStruct(q) and ClassStruct($N^k_{(i)}$). By StateStruct*(q) denote the pointer to the state structure of q and by ClassStruct*($N^k_{(i)}$) denote the pointer to the class structure of the class $N^k_{(i)}$.

Notice that to the procedure TotallyRenameBlock we pass the list of state structures and to the procedure UpdateListToSortK we pass the information about one class structure. In both procedures we have to access information about state structures of the same states. But in the procedure TotallyRenameBlock this list of state structures should be in sorted order.

At step 1 of the algorithm. the list L corresponds to the list of the classes sorted at step 1 of the third version of the minimization algorithm. Starting from the step 2 we make sure that we do not sort classes which contain the same patterns. At every step k of the algorithm we construct the new list L. By N^i_p we denote the ith name of some state p. where $k \geq i$. We should make sure that this name contains information about the $(i-1)$th name of the state p. Thus renaming of states should be done in such a way that the new name of a state p. N^k_i. contains an information about N^{i-1}_p. that means the previous name of p. The new name for any class of states should be calculated using the function f which has the following

property: f calculates a new name and there exists an inverse function for f, which can calculate the previous name of any state.

Every class structure contains the current name of a class and the marking set. The name $\Lambda^k_{(i)}$ and the marking set represent the pattern of every state belonging to the class $\Lambda^k_{(i)}$ at step k of the algorithm. After renaming, in the procedure UpdateListToSortK we are trying to update appropriate patterns. If patterns cannot be updated uniquely, this means that patterns are distinct and the class should be divided.

Example 5.3.1 Consider the automaton M_3 from Figure 5.1.1. We will show how to apply the third version of the algorithm to this automaton. Only the first step will be explained.

Suppose that after the initial partition, the classes $\Lambda^1_{(1)}$, $\Lambda^1_{(2)}$, and $\Lambda^1_{(3)}$ are given the names 1, 2 and 3 and they are put on the list to sort L. After sorting the patterns of the class $\Lambda^1_{(1)}$ the states in this class get the name 1.

Notice that in this example we do not follow the procedure TotallyRenameBlock which requires that the block of states gets a different name every time this procedure is called. But initially, when the classes of 0-indistinguishable states are sorted, it can just verify that they are 1-indistinguishable also and so there is not need to change their names. At the next steps of the algorithm, the classes of distinguishable states are sorted and so we should make sure that they all will get new names. Then the procedure UpdateListToSortK is called. Notice that only the marking set in ClassStruct($\Lambda^1_{(1)}$) is updated by this procedure, as $\Lambda^1_{(1)}$ is the only class not contained in the list to sort L. Thus after execution of this procedure we have the following class structures (notice that some elements in the marking sets have not been calculated yet which we will denote by λ).

ClassStruct($\Lambda^2_{(1)}$)

1 /* the previous name of this class */

1 /* the current name of this class */

set of states = $\{\omega\}$

marking list = (1, 1)

ClassStruct($X^1_{(2)}$)

λ /* no previous name for this class */

2 /* the current name of this class */

set of states = $\{A, B, C\}$

marking list = (λ, λ)

ClassStruct($X^1_{(3)}$)

λ /* no previous name for this class */

3 /* the current name of this class */

set of states = $\{\underline{X}, \underline{Y}, \underline{Z}\}$

marking list = (λ, λ)

Then the class $X^1_{(2)}$ is sorted and it gets the name 2. After the execution of the procedure UpdateListToSortK, the marking list of ClassStruct($X^2_{(3)}$) is updated and the class structures look as follows:

ClassStruct($X^2_{(1)}$)

1 /* the previous name of this class */

1 /* the current name of this class */

set of states = $\{\omega\}$

marking list = (1, 1)

ClassStruct($X^2_{(2)}$)

2 /* the previous name of this class */

2 /* the current name of this class */

set of states = $\{A, B, C\}$

marking list = $(2, \lambda)$

ClassStruct($\Lambda^1_{(3)}$)

λ /* no previous name for this class */

3 /* the current name of this class */

set of states = $\{\omega\}$

marking list = (λ, λ)

Then we sort $\Lambda^1_{(3)}$ and we have to rename states. The state $\underline{X}$ will get the name 31, and the states $\underline{Y}$ and $\underline{Z}$ will get the name 32. At the same time two new classes are created ($\Lambda^2_{(3)}$ and $\Lambda^2_{(4)}$). The states $\underline{X}$, $\underline{Y}$, and $\underline{Z}$ have the following state structures:

StateStruct($\underline{X}$)

transition list (StateStruct*(ω), StateStruct*($\underline{Y}$))

connection set { (StateStruct*(A), b) }

ClassStruct*($\Lambda^2_{(3)}$)

StateStruct($\underline{Y}$)

transition list (StateStruct*(C), StateStruct*(ω))

connection set { (StateStruct*(B), b), (StateStruct*($\underline{X}$), b) }

ClassStruct*($\Lambda^2_{(4)}$)

StateStruct($\underline{Z}$)

transition list (StateStruct*(B), StateStruct*(ω))

connection set { (StateStruct*(C), b) }

ClassStruct ($\Lambda^2_{(4)}$)

Consider the state $\underline{X} \in \Lambda^1_{(3)}$. As (StateStruct*($A$), b) is in the connection set of $\underline{X}$, and A is in the class $\Lambda^2_{(2)}$, in class structure of $\Lambda^2_{(2)}$ we set mark(b) = $\Lambda^2_{(3)} = \Lambda^2_{\underline{X}}$. Thus now the class $\Lambda^2_{(2)}$ has the following class structure

ClassStruct($X^2_{(2)}$)

2 /* the previous name of this class */

2 /* the current name of this class */

set of states = $\{\omega\}$

marking list = (2, 31)

State $\underline{Y}$ has two elements in the connection set. Consider StateStruct*($B.b$). As $B \in X^2_{(2)}$, in the class structure of $X^2_{(2)}$ we should set mark(b)=$X^2_{(4)} = X^2_{\underline{Y}}$. But already we have mark(b)=$X^2_{(3)}$. Note that $X^2_{(3)} \neq X^2_{(4)}$ ($31 \neq 32$), but at step 1 of the algorithm states belonging to these classes have the same name (namely 3). Thus put $X^2_{(2)}$ on the list to sort K. Then we have to consider (StateStruct*($\underline{X}$).b). Notice that $\underline{X}$ belongs to $X^2_{(3)}$ now (from $X^1_{(3)}$ the classes $X^2_{(3)}$ and $X^2_{(4)}$ were created). Thus we update class structure of $X^2_{(3)}$:

ClassStruct($X^2_{(3)}$)

3 /* the previous name of this class */

31 /* the current name of this class */

set of states = $\{\underline{X}\}$

marking list = (λ, 32)

Considering the state $\underline{Z}$ and (StateStruct(C).b) from the connection set we see that $C \in X^2_{(2)}$ which is already on the list to sort. Thus at step 2 of the algorithm we have only the class $X^2_{(2)}$ on the list to sort L.

Theorem 5.3.1 *The runtime of Algorithm 5.2.1 is* $\Theta(n^2)$.

Proof: By Theorem 5.2.1 the upper bound for this algorithm is $O(n^2)$ (Algorithm 5.3.1 makes at most as many comparisons as Algorithm 5.2.1 if constants are ignored).

To show that $\Theta(n^2)$ is the actual time complexity for this algorithm consider the automaton M_{worst} defined in Section 5.2. An example of such automaton for $n = 6$ is shown in Figure 5.2.1. To minimize this automaton. or rather to use the algorithm to verify that the automaton is minimal. only initially the whole list of patterns is sorted and then $n - 3$ different classes have to be sorted by patterns (compare Table 5.2.1 and Table 5.2.2). where n is the number of states. Notice that. even when only classes which have to be sorted are sorted. all states in these classes should be checked so at each step of the algorithm we need at least n. $n - 2$. $n - 3$. $n - 4$.... comparisons. This means that the algorithm has time complexity $\Theta(n^2)$.

The best case for this algorithm is the same as for the previous versions of the minimization algorithm. It is the case when the minimal automaton has only 2 states.

This version of the minimization algorithm requires that the names of states have a certain property. It is not enough to ensure that a new name introduced during the algorithm is distinct from all other names currently used. This name should contain the information about the previous name of a class/state. One way to accomplish that. is to find a function f which can calculate such a name. Another way is to use the element from the structure ClassStruct. representing the previous name of this structure (but then the marking list should have two fields for every every input symbol). Such an element is also needed to keep the information about the current patterns for all states (so during renaming in the procedure TotallyRenameBlock the patterns are not changed). Thus the number of distinct names needed for this version of the minimization algorithm is at most n.

CHAPTER 6

DIFFERENCE BETWEEN THE ALGORITHMS

Wood's algorithm [7] can be improved by making better use of transitivity. Given an automaton M, for some states p, q and z of this automaton, suppose that at step k of the algorithm we find out by checking the pairs (p, q) and (q, z), that the states in these pairs are k-indistinguishable. By transitivity, we can conclude that the states (p, z) are also k-indistinguishable. So there is no need to check the pair (p, z) separately. One method that efficiently uses the notion of transitivity is sorting.

Sorting procedures can also more efficiently distinguish states. Assume that at step k of the algorithm, unique names are associated with states so that sorting can be performed. Suppose that for some states p, q and z, the states in the pair (p, q) and the states in the pair (q, z) are k-distinguishable and as a result of sorting we obtain the list (p, q, z). This allows us to conclude that the states from the pair (p, z) are also k-distinguishable. This improvement applies not only to Wood's algorithm but also to Hopcroft and Ullman's algorithm [4].

The new algorithms presented in this thesis are based on sorting. All versions of the new algorithm are similar to Wood's algorithm as they have the same partition of states at every step (although the second version is the closest to Wood's algorithm). With some modifications, this algorithm can be implemented similarly to Hopcroft and Ullman's algorithm (this is the third version). Also at the same time all versions

of the new algorithm are totally different from the two existing algorithms which are based on the comparisons between pairs of states. The algorithms presented in this thesis are not based on the comparisons between pairs of states but on sorting them. We claim that it is more efficient and we will try to explain why.

The second version of the algorithm, similarly to Wood's algorithm, which has time complexity $O(n^3)$, does subsequent refinement of classes. At every step the partition given by Wood's algorithm is the same, and at every step both algorithms are checking the same states. At the same time the time complexity is better, as it is only $O(n^2)$. The reason for that is that, to get the next refinement, the states are sorted while in Wood's algorithm the states are compared pairwise giving equivalence classes. So in one step the maximum number of comparisons for the new algorithm is $n \log n$ (and if this maximum is really reached that means that the process of minimization is finished), and in Wood's algorithm it is always n^2 independently of the automaton.

It is easy to show that the second version of the algorithm is asymptotically faster than Wood's algorithm, because of the difference in time complexity. It is not the case with the algorithm due to Hopcroft and Ullman. Time complexity is the same for both algorithms.

To explain why sorting in most cases should be better for the algorithm due to Hopcroft and Ullman, consider the example where at some point you have that the following states $\{1, 2, 3, 4, 5, 6, 7, 8\}$ are not distinguished pairwise, and at some point during the algorithm we find out that after the input symbol a we distinguish states $\{1a, 2a, 3a, 4a\}$ and $\{5a, 6a, 7a, 8a\}$. So according to the algorithm due to Hopcroft and Ullman we should distinguish the pairs (1,5), (1,6), (1,7), (1,8), (2,5), (2,6), (2,7), (2,8), (3,5), (3,6), (3,7), (3,8), (4,5), (4,6), (4,7), and (4,8) so we need 16 steps. If we have more states, by dividing them into half we will need $n/2 * n/2 = n^2/4$ steps. But if we just rename states, we can sort them (n of them)

in time $n \log n$ which is asymptotically better.

There is only one case when the algorithm due to Hopcroft and Ullman can be faster doing the next refinement than algorithms using sorting methods. This is the case when we need to distinguish states $\{1\}$ and $\{2,3,4,5,6,7,8\}$ (so only one element in the new equivalence class). In that case the Hopcroft and Ullman's algorithm needs $n-1$ comparisons while, for example mergesort needs approximately $n + \log n$ comparisons.

Notice also that the *initial* division of states, when we are doing that pair by pair, in Wood's and Hopcroft and Ullman's algorithms takes n^2 steps but when we just sort states (patterns associated with states) we need only n of them as explained in *Section* 5. Whenever this initial sorting procedure is the last sorting procedure to call we have the best case for the new algorithm. So the best case for this algorithm has time complexity $O(n)$ versus $O(n^2)$ for Wood's and Hopcroft and Ullman's algorithms.

The length of any name used in the new versions of the minimization algorithm can be at most $\log n$, where n is the number of states. It can be argued that the time for each comparison of any two names should be considered when time complexity is calculated. But if we do that then also in Hopcroft and Ullman's algorithm the time required to calculate the address in the matrix should be considered.

CHAPTER 7

CONCLUDING REMARKS

7.1. Summary of Results

In this thesis the new algorithm for the minimization of a DFA is presented. The three versions of this algorithm are discussed and compared to the existing algorithms for the minimization of a DFA. It is explained why the method used in the new versions of the minimization algorithm should be more efficient than the methods used in Wood's algorithm and in the algorithm due to Hopcroft and Ullman. The main results shown in this thesis are

- A new method which can be used in the minimization of a DFA.
- The new algorithm can be used to minimize the automaton with the partion of states into more than two classes of states directly.
- It is explained which kind of existing algorithms can be improved by using the sorting method presented in this thesis.

REFERENCES

1. J. A. Brzozowski. H. Jürgensen: A model for Sequential Machine Testing and Diagnosis. *Journal of Electronic Testing: Theory and Applications* **3** (1992). 219–234.
2. F.Gésceg. I.Peák: *Algebraic Theory of Automata*. Akadémiai Kiadó. Budapest. 1987.
3. J. E. Hopcroft: An $n \log n$ Algorithm for Minimizing the States in a Finite Automaton. in: Z. Kohavi. ed.. *The Theory of Machines and Computations*. Academic Press. New York. 1971. 189–196.
4. J. E. Hopcroft. J. D. Ullman: *Introduction to Automata Theory. Languages and Computations*. Addison-Wesley, New Jersey . March 1979.
5. P. M. B. Vitányi. L. Meertens. Big Omega Versus the Wild Functions. *SIGACT News*. Volume 16. number 4. 1985. 56-59.
6. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*. Eindhoven University of Technology, The Netherlands. 1995.
7. D. Wood:. *Theory of Computation*. John Wiley & Sons. New York. 1987.