Department of Computer Science
Series of Publications A
Report A-2004-9

On Minimality and Size Reduction of One-Tape and Multitape Finite Automata

Hellis Tamm

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium CK112, Department of Computer Science, on December 10th, 2004, at 12 o'clock noon.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2 b)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 51120

ISSN 1238-8645
ISBN 952-10-2195-0 (paperback)
ISBN 952-10-2196-9 (PDF)
Computing Reviews (1998) Classification: F.1.1, F.2.2
Helsinki 2004
Helsinki University Printing House

On Minimality and Size Reduction of One-Tape and Multitape Finite Automata

Hellis Tamm

Department of Computer Science
P.O. Box 68, FIN-00014 University of Helsinki, Finland
hellis.tamm@helsinki.fi

PhD Thesis, Series of Publications A, Report A-2004-9
Helsinki, November 2004, 80 pages
ISSN 1238-8645
ISBN 952-10-2195-0 (paperback)
ISBN 952-10-2196-9 (PDF)

Abstract

In this thesis, we consider minimality and size reduction issues of one-tape and multitape automata. Although the topic of minimization of one-tape automata has been widely studied for many years, it seems that some issues have not gained attention. One of these issues concerns finding specific conditions on automata that imply their minimality in the class of nondeterministic finite automata (NFA) accepting the same language.

Using the theory of NFA minimization developed by Kameda and Weiner in 1970, we show that any bideterministic automaton (that is, a deterministic automaton with its reversal also being deterministic) is a unique minimal automaton among all NFA accepting its language. In addition to the minimality in regard to the number of states, we also show its minimality in the number of transitions. Using the same theory of Kameda and Weiner, we also obtain a more general minimality result. We specify a set of sufficient conditions under which a minimal deterministic automaton (DFA) accepting some language or the reversal of the minimal DFA of the reversal language is a minimal NFA of the language.

We also consider multitape bideterministic automata and show by a counterexample that such automata are not necessarily minimal. However, given a set of accepting computations of a bideterministic multitape automaton, we show that this automaton is a unique minimal automaton with this set of accepting computations.

We have also developed a polynomial-time algorithm to reduce the size of (one-way) multitape automata. This algorithm is based on simple language-preserving automata transformations that change the order in which transitions involving different tapes occur in the automaton graph and merge suitable states together. We present an example of a family of automata on which the reduction algorithm works well.

Finally, we apply the multitape-automata size-reduction algorithm along with the DFA minimization procedure to the two-way multitape automata appearing in a string-manipulating database system. We have implemented software to empirically evaluate our size-reduction algorithm on these automata. We have done experiments with automata corresponding to a set of string predicates defining several different string properties. Good results of these experiments suggest the usefulness of this approach on reducing the size of the automata that appear in this system.

Computing Reviews (1998) Categories and Subject Descriptors:
F.1.1 [Theory of Computation]: Computation by Abstract Devices – Models of Computation
F.2.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity – Nonnumerical Algorithms and Problems

General Terms:
Theory, Algorithms

Additional Key Words and Phrases:
Minimal Automata, Bideterministic Automata, Multitape Automata

Acknowledgements

I am very grateful to my advisor Professor Esko Ukkonen. His scientific expertise has made it possible for him to give me directions and advice when needed, and still he has let me to work on problems which are somewhat less related to his other current research. I am thankful for his guidance, patience, optimism and support that has made it possible for me to finish this thesis.

Concerning the early stage of this work, I am very grateful to Matti Nykänen and Raul Hakli for our discussions and joint publications which have provided me the necessary platform on which this thesis could be built.

I also wish to thank Dr. Tero Harju and Professor Seppo Sippu for their useful comments on the manuscript of this thesis that improved the presentation, and I thank Marina Kurtén for correcting the language.

I am grateful for the very good working conditions I have had in the Department of Computer Science of the University of Helsinki. I also appreciate the opportunity of being a member of the From Data to Knowledge (FDK) research unit. The Academy of Finland has provided me financial support.

However, I am most thankful to God and to my family and friends who have given so much to me. Especially, I am grateful for the prayers and support of my parents Maila and Feliks. I also thank my friend Ruth for her longtime friendship and the things I have learned from her. Finally, I am very grateful to my husband Toomas for his support, encouragement and help on various matters. Thank you for being there for me.

Contents

Chapter 1

Introduction

During its fifty years of development, automata theory has become one of the foundations of computer science. Automata theory has many branches with different kinds of automata models introduced over these years, with many applications such as programming languages, compilers (e.g. [1]), verification systems etc. Some other recent applications in algebra, combinatorics, and image manipulation are considered in [25]. Although being a well-studied part of computer science, automata theory is still an attractive research area today with a wide range of topics.

One important as well as interesting topic in automata theory is *minimization* and *size reduction* of automata. If not specified otherwise, the problem of minimization is usually understood as finding an automaton with the minimum number of states, which accepts the language of a given automaton. Such an automaton is said to be of minimal size. There is much research done on the subject of minimization of one-tape automata. Here, quite often, the minimization is understood in even narrower context, namely as finding a deterministic finite automaton (DFA) with the smallest number of states, accepting the given language. The minimization of a DFA can be done efficiently based on the Myhill-Nerode theorem [19, Theorem 3.9]. This theorem specifies the largest right-invariant equivalence relation on the states of a DFA, indicating which states are *equivalent* and thus can be merged, resulting in a minimal DFA. For every regular language there is a unique (up to isomorphism) minimal DFA recognizing it. Many DFA minimization algorithms have been proposed, of which Hopcroft's algorithm [17] has the best running time of $O(n \log n)$ where n is the number of automaton states. A broad overview of DFA minimization algorithms is presented in [35].

However, the minimal DFA is not necessarily a minimal automaton accepting its language. In fact, the size of the minimal DFA of a given

language can be exponentially larger than the size of a minimal nondeterministic finite automaton (NFA) of that language. However, finding a minimal NFA for a given language is a more difficult problem. The decision problem of finding a minimal NFA when given a DFA of a language is a PSPACE-complete problem [23]. Moreover, contrary to the uniqueness property of the minimal DFA, in the class of NFAs there may exist more than one automaton of minimal size accepting the given language. The problem of NFA minimization has been a topic of several papers, for example, [3, 24, 27, 31]. Because of the difficulty of the problem, efficient algorithms to obtain NFAs that are reduced in some specific manner, instead of strictly minimal ones, can be useful. For example, the same approach as in DFA minimization, namely, finding the largest right-invariant equivalence of the states of an NFA and then merging the equivalent states, is used for size reduction of NFAs in [20] and [21]. The latter also considers the left-invariant equivalence of the states for merging the states of an NFA. By examples, good results are obtained in [21] if both of these equivalences are used to reduce the size of an NFA but the problem of how to combine them to get the best results is open. Another method for reducing the size of NFAs using preorders instead of equivalence relations is considered in [5]. Both of these methods, that is, the one using equivalences and the other using preorders, are considered in [22] which proposes fast algorithms for these methods.

Other means to obtain minimal NFAs such as specifying lower bounds for the size of a minimal NFA in special cases (for example, [8]), or determining other conditions under which an automaton is a minimal NFA, can also be useful. In this thesis, we are interested in finding specific conditions on automata which imply their minimality among all NFAs accepting their languages. Efficiently testable conditions can be an easy way to prove that the automaton in question is minimal. We address this problem in this thesis by showing that a special class of automata called *bideterministic* automata are minimal among NFAs. We also present a more general, although technical, set of sufficient conditions for an automaton to be a minimal NFA.

The second subject of this thesis is size reduction of multitape automata. Multitape automata, or more specifically, one-way multitape nonwriting finite automata, introduced by Rabin and Scott in [30], clearly are a more difficult area of research than one-tape automata. Contrary to the one-tape case, not all nondeterministic multitape automata have an equivalent deterministic counterpart, that is, a deterministic multitape automaton that accepts the same language. In other words, the nondeterministic multitape

automaton model has a bigger language-defining power than the deterministic multitape model. Also, many decision problems that are solvable for one-tape automata are unsolvable for nondeterministic multitape automata. One example of this kind of an undecidable problem is the equivalence of nondeterministic automata [11]. The equivalence of deterministic multitape automata has been found to be decidable [16] after being an open problem for a long time. Research has been done on finding a system of transformations with the property that for any two equivalent deterministic multitape automata there exists a sequence of transformations in this system that transforms one automaton to another [26]. However, we are not aware of any attempt to find a minimization or even a size-reduction procedure for multitape automata.

In this thesis, we consider a modification of the Rabin-Scott (one-way) multitape automaton model [30]. In the Rabin-Scott model, every automaton state is associated with a certain tape and all transitions leaving the same state read a symbol (indicated by the transition label) from that tape. In our model, no such state-tape association is made but different transitions leaving the same state can involve different tapes. A transition label in this model indicates the tape involved in this transition and the symbol read from that tape. We present a size-reduction algorithm for automata in this model that is based on simple automata transformations. While this algorithm does not pretend to reduce the size of every reducible automaton, there are cases in which it seems to work nicely.

The motivation for the multitape-automata size-reduction algorithm came from the development of the string-manipulation database system described in [9, 13, 14, 15]. This system involves a string-manipulating extension where string predicates can be expressed using a specific language called Alignment Declaration Language. These string predicates in the form of alignment declarations are then compiled into two-way multitape automata which are further transformed into executable programs. The model for presenting these two-way multitape automata is very similar to the modified Rabin-Scott model mentioned above. The main difference is that in addition to the normal transitions that read the automaton tapes, there are transitions that correspond to the tape head movements to the left or right.

Our interest (in this issue, in regard to this thesis) is to try to reduce the size of these two-way automata. Our approach is as follows. As our model for presenting these two-way multitape automata is very similar to the one-way multitape automaton model mentioned above (detailed descriptions are presented in Section 2.2), we would like to use the one-way multitape

automata size-reduction algorithm also for reducing the size of our two-way multitape automata.

For this reason, we define a way to interpret a two-way multitape automaton as if it was a one-way multitape automaton instead. More specifically, we expand the alphabet of the one-way automaton so that it includes also the special symbols that denote the tape movements in the transitions of the two-way automaton. Then we can apply the size-reduction algorithm to the one-way multitape automaton obtained this way. When we interpret the resulting automaton back as a two-way automaton, it is equivalent to the original two-way automaton because the transformations performed on the corresponding one-way automaton are equivalence preserving. In addition, we also define a way to interpret a one-way multitape automaton as if it was a (one-way) one-tape automaton instead, by a similar alphabet expansion as above. This allows us to apply algorithms developed for one-tape automata, on one-way multitape automata.

Combining these two levels of interpretations between the automaton models, we use a method to reduce the size of a given two-way multitape automaton, which involves two algorithms. Our (one-way) multitape-automata size-reduction algorithm mentioned above, along with the (one-way one-tape) DFA-minimization procedure, is used in this method, with appropriate interpretations of one automaton model into another. We apply these two algorithms in alternate manner until no more reduction of the automaton size is achieved. The language of the resulting two-way multitape automaton is the same as the language of the original automaton. This approach seems to work well on reducing the size of the automata corresponding to alignment declarations, as the experimental results in Chapter 6 suggest.

1.1 Contributions of this work

Using the theory of NFA minimization developed by Kameda and Weiner in [24], we show that any bideterministic automaton (that is, a deterministic automaton with its reversal also being deterministic) is a minimal automaton among all NFAs accepting its language. We also show that any non-deterministic (that is, not deterministic) automaton equivalent to a bideterministic automaton has more states than the latter one. This result along with the earlier-known and easily-seen fact that a bideterministic automaton is the unique minimal DFA of its language (observed, for example, by [2] and [29]) yields the result that a bideterministic automaton is a unique minimal automaton accepting the given language. In addition

to the minimality in regard to the number of states, we also show that a bideterministic automaton has a minimal number of transitions.

Using the same theory of Kameda and Weiner, we also obtain a more general minimality result. We specify a set of sufficient conditions under which a minimal DFA accepting some language, or the reversal of the minimal DFA of the reversal language, is a minimal NFA of the language. Although technical, these conditions are not difficult to test and provide a simple way to prove the minimality of automata. Actually, any bideterministic automaton meets these conditions, so the minimality of bideterministic automata can be obtained using this result as well.

We also consider multitape bideterministic automata and show by a counterexample that such automata are not necessarily minimal. However, given a set of accepting computations of a bideterministic multitape automaton, we show that this automaton is a unique minimal automaton with this set of accepting computations.

We have also developed a polynomial-time algorithm to reduce the size of (one-way) multitape automata. This algorithm is based on four simple language-preserving automaton transformations that change the order in which transitions involving different tapes appear in the automaton graph and merge suitable states together. We have specified a set of sufficient conditions concerning certain transitions and paths in the automaton graph; if these conditions hold then the transformations reduce the automaton size by a specified amount. Also, these transformations eliminate at least the same number of transitions from the automaton. We present an example of a family of automata on which the reduction algorithm works well.

Finally, we apply the multitape-automata size-reduction algorithm together with the DFA-minimization procedure to the two-way multitape automata appearing in the string-manipulating database system of [9]. We have implemented software to empirically evaluate our size-reduction algorithm on these automata. We have done experiments with automata corresponding to a set of string predicates defining several different string properties. Good results of these experiments suggest the usefulness of this approach on reducing the size of the automata that appear in this system.

Most of the results concerning the minimality issues of one-tape automata have appeared earlier in the conference paper [33] and in the journal version of that article [34].

1.2 Overview of the thesis

This thesis has the following structure. In Chapter 2 we give the definitions of one-tape and multitape automata models that we use and show how to interpret automata given in a certain model in other models. In Chapter 3 we present the main results of the NFA minimization theory of Kameda and Weiner [24], and based on this theory, prove the minimality of bideterministic automata as well as the theorem specifying a set of more general conditions guaranteeing the minimality of certain automata. The results concerning bideterministic multitape automata are presented in Chapter 4. We develop the multitape-automata size-reduction algorithm in Chapter 5, and in Chapter 6 we apply the reduction algorithm along with the DFA-minimization procedure to the automata appearing in a string-manipulation database system. Finally, the conclusions are presented in Chapter 7.

Chapter 2

One-tape and multitape finite automata

In this chapter we present the definitions of one-tape and multitape finite automata and related concepts. First we discuss the well known one-tape automaton model. From the models of multitape automata, we consider the classical *Rabin-Scott model* and its modification that we call the *mixed-state model*. Both of these are one-way automaton models. We also consider a two-way version of the mixed-state model, motivated by the development of the string database system of [9]. For the purposes of applying the techniques developed for one-tape or one-way multitape automata on two-way multitape automata in Chapter 6, we describe here a way to interpret two-way multitape automata as one-way multitape automata, and one-way multitape automata as one-tape automata.

2.1 One-tape automata

One of the most well known automaton models involves one *tape* and one *head* to scan that tape, moving in one direction only. The symbols on the tape are scanned from left to right, one at a time, starting from the first symbol and ending with the last symbol on the tape. When the automaton is in some of its *states*, it reads the next symbol on the tape and, depending on the state-symbol pair, goes to the next state. In the beginning, the automaton is in one of its *initial states*. The automaton *accepts* the string on the tape if, after reading the last symbol on the tape, the automaton is in one of its *accepting states*.

Formally, a one-tape finite automaton is a quintuple $A = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of *states*, Σ is the *input alphabet*, $\delta : Q \times \Sigma \to 2^Q$ is the

transition function, $I \subseteq Q$ is the set of *initial states* and $F \subseteq Q$ is the set of *final states*. An automaton A is *deterministic* (DFA) if it has a unique initial state and if for every $q \in Q$ and every $a \in \Sigma$, $|\delta(q,a)| \leq 1$. The general case of finite automata is *nondeterministic* (NFA). The *reversal* of an automaton A is the automaton $A^R = (Q, \Sigma, \delta^R, F, I)$ where $\delta^R(p,a) = \{q \mid p \in \delta(q,a)\}$ for all $p \in Q$ and $a \in \Sigma$. An automaton A is called *bideterministic* if both A and its reversal automaton A^R are deterministic.

The *empty string* is denoted by ϵ. For any string $x = x_1...x_k$, where $x_i \in \Sigma$ for $i = 1, ..., k$, we denote by x^R the *reversal* of x which is the string $x_k...x_1$.

We define the *extended transition function* $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ so that $\hat{\delta}(q,\epsilon) = \{q\}$ and $\hat{\delta}(q,xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p,a)$ for all $q \in Q$, $x \in \Sigma^*$ and $a \in \Sigma$. A string $x \in \Sigma^*$ is *accepted* by A if there exists $q_0 \in I$ such that $\hat{\delta}(q_0,x) \cap F \neq \emptyset$. The set $L(A) = \{x \mid \bigcup_{q \in I} \hat{\delta}(q,x) \cap F \neq \emptyset\}$ is called the *language* accepted by A. The *reversal* of a language L, denoted by L^R, is the set of the reversals of all the strings belonging to L. A language accepted by a bideterministic automaton is a *bideterministic language*.

A *minimal* automaton is an automaton with the smallest number of states among all automata that accept the given language. A state q of A is *useful* if $L((Q, \Sigma, \delta, I, \{q\})) \neq \emptyset$ and $L((Q, \Sigma, \delta, \{q\}, F)) \neq \emptyset$. Two states q_i and q_j of A are *equivalent* if $L((Q, \Sigma, \delta, \{q_i\}, F)) = L((Q, \Sigma, \delta, \{q_j\}, F))$. Using the Myhill-Nerode theorem [19, Theorem 3.9] it can be proven that a deterministic automaton is minimal among all DFAs accepting the same language if and only if all of its states are useful and no two states of it are equivalent. Two automata are equivalent if they accept the same language. Given an automaton A, using the well-known operation of the *subset construction*, we obtain an equivalent deterministic automaton $D(A) = (Q', \Sigma, \delta', \{q'\}, F')$ [18, Section 2.3.5], [19, Theorem 2.1]. We also call this operation *determinization*. The automaton $D(A)$ consists of only *useful* states, that is, the states that appear on some path from the initial state to a final state of the automaton. However, it is not necessarily the minimal DFA for $L(A)$.

Sometimes a stricter notion of determinism of an automaton than the definition given above is used by requiring that for all $q \in Q$ and $a \in \Sigma$, $|\delta(q,a)| = 1$ (instead of $|\delta(q,a)| \leq 1$). This implies that some deterministic automata must have a so-called *dead state* $q_\emptyset$ such that $q_\emptyset \notin F$ and $\delta(q_\emptyset, a) = q_\emptyset$ for all $a \in \Sigma$. With this notion of determinism, the class of bideterministic automata is smaller when compared to the class of bideterministic automata obtained by using the definition of determinism as above.

A deterministic automaton with a dead state $q_\emptyset$ cannot be bideterministic as there must be some $a \in \Sigma$ for which $|\delta^R(q_\emptyset, a)| > 1$. For example, while the language L_2^* of Example 3.3 in Section 3.2 is bideterministic by our definition of determinism, it is not bideterministic by this stricter notion of determinism. An example of a language that is bideterministic according to both definitions is the language $L((\mathbf{0}+\mathbf{1})((\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1}))^*)$ of Example 3.4.

Sometimes we allow an automaton to have transitions on the empty string ϵ. Then the transition function of the automaton is $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$. We call such an automaton an ϵ-NFA.

2.2 Multitape automata

There are several models of multitape automata that have been presented over the years. The most known are perhaps the Rabin-Scott model introduced in [30], the Elgot-Mezei model [6], and the read-only one-way Turing machine model. An overview of these models was given in [7]. Here we consider two models of multitape automata: first, the Rabin-Scott model, and second, a modified version of that model which we call the mixed-state model. Both are one-way models but we also present a two-way version of the latter model.

2.2.1 The Rabin-Scott model

In the *Rabin-Scott model* a machine has n tapes and a scanning head for every tape. The beginning and the end of each tape are indicated, respectively, by the *left endmarker* [and the *right endmarker*]. These two are special symbols not belonging into the input alphabet of the automaton. Initially, the machine is in one of the initial states, with its heads placed on the left endmarkers for all tapes. The first symbol the machine reads on each tape is the left endmarker. The reading of the tapes is done so that only one tape is read at a time. For this reason every state of the machine is associated with one of the tapes. When the machine is in some state it reads the next symbol on the tape associated with that state, and depending on the symbol read goes into the next state indicated by the state-symbol pair. The n-tuple of tapes is accepted by the machine if it is in a final state after reading the right endmarker on all of its tapes.[1]

[1]This definition of acceptance is different from the one in [30] where the acceptance criterion is that the machine is in a final state after reading the right endmarker on one of its tapes. Also, the original model described only deterministic automata and did not have the left endmarker. While the left endmarker is not important in this model, the

Let us assume that a function $tape : Q \to \{1, ..., n\}$ associates every state of the automaton with a certain tape. Now, formally, an n-tape automaton is given by a six-tuple $(Q, tape, \Sigma, \delta, I, F)$ where Q is a finite set of states with a partition into the sets $Q_1, ..., Q_n$ so that $Q_i = \{q \in Q | tape(q) = i\}$ for $i = 1, ..., n$, Σ is the input alphabet, $\delta : Q \times (\Sigma \cup \{[,]\}) \to 2^Q$ is the transition function, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states. If for some $q_1, q_2 \in Q$ and $a \in \Sigma \cup \{[,]\}$, $q_2 \in \delta(q_1, a)$, then we say that there is a transition from q_1 to q_2 with label a. As in the one-tape case, an automaton is deterministic if $|I| = 1$ and if for all $q \in Q$ and $a \in \Sigma \cup \{[,]\}$, $|\delta(q, a)| \leq 1$.

In the following we consider a multitape automaton as a directed graph in the usual way. In an automaton graph we define an *accepting path* to be a path that leads from an initial state into a final state. Given a transition from a state q_1 with a label a, we define the *indexed label* of that transition to be $a_{tape(q_1)}$. We usually use indexed labels for labelling transitions when drawing automata graphs. We define an *accepting computation* of an automaton to be a string formed by concatenating the indexed labels of all transitions that appear on an accepting path. We denote the set of all accepting computations of an automaton A by $C(A)$. We say that an n-tuple $(w_1, ..., w_n)$, where $w_i \in \Sigma^*$ for $i = 1, ..., n$, is *accepted* by A if there exists an accepting computation u of A such that for all $i \in \{1, ..., n\}$, w_i is a string obtained from u by removing from u the symbols $[_i$, $]_i$ and all such symbols a_j where $i \neq j$, $a \in \Sigma \cup \{[,]\}$, and replacing the remaining indexed label symbols in u by their corresponding labels (without indexes). The set of all n-tuples accepted by A is the *language* of A, denoted by $L(A)$. As in the one-tape case, two automata are equivalent if they accept the same language.

Actually, the left endmarker is not necessary in this model, since it is a one-way model. The role of the right endmarker is important, however, illustrated by the following example from [12, Section 3.1]. Consider the language $\{(a, \epsilon), (\epsilon, b)\}$ accepted by a two-tape automaton with the alphabet $\Sigma = \{a, b\}$. It is easy to see that there are no deterministic Rabin-Scott two-tape automata without the right endmarker which accept this language, whereas this language is accepted by such an automaton with the right endmarker, as shown in Figure 2.1.

---

role of the right endmarker is discussed at the end of this section.

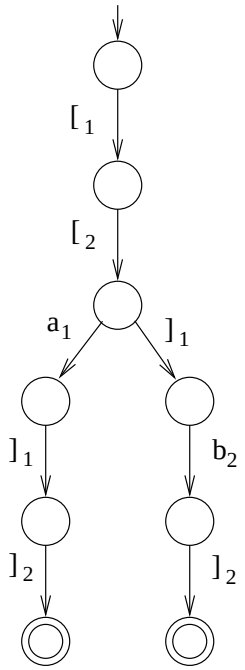

Figure 2.1: A two-tape automaton accepting the language $\{(a, \epsilon), (\epsilon, b)\}$. The indexes of the labels at the transitions indicate the tapes that are read.

2.2.2 Mixed-state model

In the Rabin-Scott model every state of an automaton is associated with a certain tape of that automaton and all transitions that leave the state concern the same tape. In the following we consider another automaton model which may be viewed as a modified version of the Rabin-Scott model. In this model, no state of an automaton is associated with any tape, therefore different transitions leaving any state may concern different tapes. Every transition label also indicates the tape which is involved in that transition, in addition to the symbol read on that tape. Also, the endmarkers are not used in this model. We call this model a *mixed-state model* and describe it more formally in the following. A similar, but more restricted automaton model was considered in [36].

An n-tape automaton in the mixed-state model is given by a quintuple $(Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, Σ is the input alphabet, $\delta : Q \times \Sigma_{\{1,...,n\}} \to 2^Q$ is the transition function where $\Sigma_{\{1,...,n\}} = \{a_i \mid a \in \Sigma, i \in \{1, ..., n\}\}$, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states. If for some $q_1, q_2 \in Q$ and $a_i \in \Sigma_{\{1,...,n\}}$, $q_2 \in \delta(q_1, a_i)$, then we say that there is a transition from q_1 to q_2 with label a_i, that is, with symbol a involving tape i. This transition is denoted by (q_1, a_i, q_2) or $q_1 \xrightarrow{a_i} q_2$. In case we need to use indexes to denote an alphabet symbol itself, we put the symbol in brackets like, for example, in $(a_k)_i$ where $a_k \in \Sigma$ and $i \in \{1, ..., n\}$. The number of outgoing and incoming transitions of a state q is denoted by $outdegree(q)$ and $indegree(q)$, respectively. Transition labels in this model are closely related to the indexed labels of transitions in the Rabin-Scott model as presented in Section 2.2.1.

Similarly to Section 2.2.1, an accepting computation of an automaton in the mixed-state model is a string formed by concatenating the labels of all transitions that appear on some path in the automaton graph that goes from an initial state to a final state. Also, we denote the set of all accepting computations of an automaton A by $C(A)$. And similarly, an n-tuple $(w_1, ..., w_n)$ where $w_i \in \Sigma^*$ for $i = 1, ..., n$, is accepted by A if there exists an accepting computation u of A such that for all $i \in \{1, ..., n\}$, w_i is a string obtained from u by removing from u all symbols a_j such that $i \neq j$, $a \in \Sigma$, and discarding the tape indexes of all remaining symbols in u. We may say that the accepting computation u *produces* the n-tuple $(w_1, ..., w_n)$. As before, the set of all n-tuples accepted by A is the language of A, denoted by $L(A)$.

Similarly to the one-tape automaton model, sometimes a multitape automaton in the mixed-state model can have transitions on the empty string ϵ. Then the transition function of the automaton is $\delta : Q \times (\Sigma_{\{1,...,n\}} \cup \{\epsilon\}) \to$

2^Q.

2.2.3 Two-way mixed-state model

In the application part of this thesis, in Chapter 6, we deal with multitape automata in a mixed-state *two-way* model, that is, a model in which the automaton tapes can be scanned in two directions: from the left to the right as well as from the right to the left. Also, these automata can have endmarkers [and] as well as ϵ-transitions.

We can see this n-tape automaton model in the following way. There is a window whose width is one symbol and height is n symbols, so that exactly one symbol of each tape shows through that window at any given time. We call the position of the showing symbol the *current position* for the corresponding tape, the symbol itself is called the *current symbol*. Initially, the current symbols for all tapes are their left endmarkers. If we want to read the next symbol from any tape, we move that tape *left* with respect to the window. And if we want to read the previous symbol from a tape, we move that tape *right*. These tape movements are indicated in the automaton as transitions with the labels L_i and R_i where L and R are special symbols not belonging to the alphabet of the automaton, and i is the tape involved.

An n-tape automaton in the two-way mixed-state model is given by a quintuple $(Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, Σ is the input alphabet, $\delta : Q \times (\Sigma'_{\{1,...,n\}} \cup \{\epsilon\}) \to 2^Q$ is the transition function where $\Sigma' = \Sigma \cup \{[,]\} \cup \{L, R\}$ and $\Sigma'_{\{1,...,n\}} = \{a_i \mid a \in \Sigma', i \in \{1, ..., n\}\}$, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states.

Let u be a string formed by concatenating the labels of all transitions that appear on some path in the automaton graph that goes from an initial state to a final state. We consider u to be an accepting computation if there exists an n-tuple $(w_1, ..., w_n)$ where $w_i \in \Sigma^*$ for $i = 1, ..., n$, such that for all $i \in \{1, ..., n\}$, w_i is in *compliance* with u in the sense that if we read u from the left to the right one symbol at a time then, on seeing L_i we read the next symbol on w_i, and on seeing R_i we read the previous symbol on w_i, and on seeing any c_i where $c \in \Sigma \cup \{[,]\}$ the symbol currently read from w_i is c. In this case, the n-tuple $(w_1, ..., w_n)$ is accepted by the automaton.

As before, the set of all n-tuples accepted by A is the language of A, denoted by $L(A)$.

2.3 Interpreting automata in other models

Later in this thesis we will find it useful to interpret a two-way multitape automaton as if it were a one-way multitape automaton that accepts a superset of the computations of the original automaton. More specifically, we expand the alphabet of the one-way automaton so that it also includes the special symbols that denote the tape movements in the transitions of the two-way automaton. Then we can apply the techniques developed for one-way multitape automata to this automaton.

Furthermore, we can interpret a one-way multitape automaton as a (one-way) one-tape automaton by a similar expansion of the tape alphabet, and apply one-tape automata methods to this automaton. In this section we show how these kinds of interpretations can be done.

Let $A = (Q, \Sigma, \delta, I, F)$ be an n-tape automaton in the two-way mixed-state model. The corresponding one-way mixed-state n-tape automaton $A' = (Q, \Sigma \cup \{L, R, [,]\}, \delta, I, F)$ is obtained from A by interpreting the symbols L, R, [and] in the transition labels of A as if they were just symbols from the input alphabet.

And conversely, having a one-way mixed-state n-tape automaton A' with an input alphabet $\Sigma \cup \{L, R, [,]\}$, its two-way counterpart A is obtained by interpreting transitions labelled by L_i, R_i, $[_i$ and $]_i$ where $i \in \{1, ..., n\}$ in the way they are interpreted in the two-way mixed-state model.

Proposition 2.1 *Let A be a two-way n-tape automaton and let A' be the corresponding one-way automaton as defined above. Let A'_1 be a one-way n-tape automaton equivalent to A', with A_1 as its two-way counterpart. Then $L(A) = L(A_1)$.*

Proof. Assume any tuple $(w_1, ..., w_n) \in L(A)$. Then there exists an accepting computation u of A such that for all $i \in \{1, ..., n\}$, w_i is in compliance with u in the sense described at the end of Section 2.2.3. Obviously, there exists an accepting computation u' of A' that is equal to u. Let $(w'_1, ..., w'_n)$ be produced by u'. As $L(A') = L(A'_1)$ then there exists an accepting computation u'_1 of A'_1 that produces the same n-tuple $(w'_1, ..., w'_n)$. From this we infer that there exists an accepting computation $u_1 = u'_1$ of A_1 such that for all $i \in \{1, ..., n\}$, w_i is in compliance with u_1. That means $(w_1, ..., w_n) \in L(A_1)$. Similarly, it can be reasoned that if $(w_1, ..., w_n) \in L(A_1)$ then $(w_1, ..., w_n) \in L(A)$. Thus, $L(A) = L(A_1)$. □

By Proposition 2.1, if we first interpret a two-way multitape automaton A as a one-way multitape automaton A', and then apply any language-

preserving transformation on A', and interpret the resulting automaton back as a two-way automaton, then the final automaton accepts the same language as the original automaton.

Note, however, that the sets of accepting computations $C(A)$ and $C(A')$ may be different. The reason is that in a two-way automaton not necessarily every path from an initial state to a final state defines an accepting computation, differently from the one-way case. This is due to the property of the two-way automaton that its tapes can be read in both directions. When reading a tape first forwards and then backwards, the choice of transitions included in the path of an accepting computation is limited by the tape symbols already read forwards. Thus, $C(A) \subseteq C(A')$.

Now, let $A' = (Q, \Sigma, \delta, I, F)$ be an n-tape automaton in the one-way mixed-state model. We obtain the corresponding one-tape automaton $A'' = (Q, \Sigma_{\{1,...,n\}}, \delta, I, F)$ from A' by interpreting the transition labels of A' as if they were just symbols from the input alphabet $\Sigma_{\{1,...,n\}}$ that are read from a single tape. And conversely, having a one-tape automaton A'' with an input alphabet $\Sigma_{\{1,...,n\}}$, its n-tape counterpart is obtained by interpreting each of its transitions labelled by any $a_i \in \Sigma_{\{1,...,n\}}$ where $a \in \Sigma$ and $i \in \{1, ..., n\}$ as a transition with a symbol a involving a tape i.

Proposition 2.2 *Let A' be an n-tape automaton and A'' be the corresponding one-tape automaton as defined above. Let A_1'' be a one-tape automaton equivalent to A'', with A_1' as its n-tape counterpart. Then $L(A') = L(A_1')$.*

Proof. The accepting computations of a one-way n-tape automaton and its corresponding one-tape automaton coincide, so $C(A') = C(A'')$. Furthermore, for a one-tape automaton, the set of accepting computations is equal to the language of that automaton, so $C(A'') = L(A'')$. In the same way, $L(A_1'') = C(A_1'') = C(A_1')$. As $L(A'') = L(A_1'')$, we get $C(A') = C(A_1')$ which implies $L(A') = L(A_1')$. □

By Proposition 2.2, if we interpret a one-way multitape automaton A' as a one-tape automaton A'', apply any algorithm on A'' that maintains the language accepted by it (for example, determinization or DFA minimization), and interpret the resulting automaton back as a multitape automaton, then the final automaton accepts the same language as the original automaton. Here, the sets of accepting computations $C(A')$ and $C(A'')$ of corresponding multitape and one-tape automata are the same.

Propositions 2.1 and 2.2 allow us to interpret a two-way multitape automaton either as a one-way multitape automaton or a one-tape automaton

and apply a combination of language-preserving algorithms that are developed for either one-way multitape automata or one-tape automata without changing the language accepted by the original automaton. This kind of approach seems to be useful for reducing the size of the two-way automata considered in Chapter 6. On these automata we will apply the one-way multitape automata size reduction algorithm developed in Chapter 5 along with the one-tape DFA minimization procedure, with apparently good results.

Chapter 3

Minimality of one-tape automata

In this chapter we present some sufficient conditions for an automaton to be a minimal NFA. Our results are based on an NFA minimization theory developed by Kameda and Weiner [24]. After a brief overview of their results, we first show that any bideterministic automaton is the unique minimal automaton accepting its language. We also present a more general theorem, specifying a set of conditions under which the minimal DFA of a given language or the reversal of the minimal DFA of the reversal language is a minimal NFA accepting the given language. In fact, the minimality of a bideterministic automaton can be concluded from that result as well. Most of the results presented in this chapter have appeared in [34].

3.1 NFA minimization of Kameda and Weiner

Kameda and Weiner [24] have developed a theory for attacking the problem of minimization of nondeterministic automata. In the following, we present some definitions and results from this theory that we will need to prove our results.

Let $A = (Q, \Sigma, \delta, I, F)$ be an automaton, let B be the determinized automaton $D(A) = (Q', \Sigma, \delta', q', F')$ and let C be the determinized automaton $D(A^R) = (Q'', \Sigma, \delta'', q'', F'')$. As B and C are results of the subset construction applied on the set of states Q of A, both Q' and Q'' consist of subsets of Q.

Definition 3.1 (Kameda and Weiner [24, Definition 7]). *The* states map (SM) *of A is a matrix which contains a row for each nonempty state of B, and a column for each nonempty state of C. The (i, j) entry contains $q'_i \cap q''_j$ (or is blank if $q'_i \cap q''_j = \emptyset$) where q'_i is the i-th element of Q' and q''_j*

is the j-th element of Q''. The elementary automaton matrix (EAM) *of A is obtained from the SM of A by replacing each nonblank entry by a 1. Its (i, j) element is denoted by e_{ij}.*

Theorem 3.1 (Kameda and Weiner [24, Theorem 3]).
$$L((Q', \Sigma, \delta', q'_i, F')) = \bigcup_{j|e_{ij}=1} \{x^R \mid x \in L((Q'', \Sigma, \delta'', q'', q''_j))\}.$$

It is observed in [24] that, according to Theorem 3.1, any states of B that have an identical pattern of 1s and blanks in the corresponding rows of the EAM of A, can be merged (by union, as the equivalent states). Also, because the definitions of B and C are symmetric, any states of C that have the same pattern of 1s and blanks in the corresponding columns, can be merged. These observations imply that two states of B (C) having the same pattern of blank entries in the corresponding rows (columns) of the SM of A can be merged. Rows (columns) of the SM with the same pattern of blank entries are called *equivalent*.

Definition 3.2 (Kameda and Weiner [24, Definitions 8 and 10]). *The* reduced states map (RSM) *of A is obtained from the SM of A by merging all equivalent rows and columns. The merging of two rows (columns) means that they are replaced by a new row (column), the entries of which are the unions of the entries of the corresponding columns (rows). The* reduced automaton matrix (RAM) *of A is formed from the RSM of A by replacing each nonblank entry with a 1.*

Let $\hat{B}$ be the minimal DFA for $L(A)$, obtained from B by merging by union the equivalent states, and let $\hat{C}$ be the minimal DFA, similarly obtained from C, for $L(C) = L(A)^R$.

Lemma 3.1 (Kameda and Weiner [24, Lemma 3]). *The RSM of A can be obtained from $\hat{B}$ and $\hat{C}$ in the same manner as the SM of A is obtained from B and C.*

Theorem 3.2 (Kameda and Weiner [24, Theorem 4]). *Equivalent automata have a RAM that is unique up to permutation of the rows and columns.*

Definition 3.3 (Kameda and Weiner [24, Definitions 11–13]). *Given an EAM or RAM, if all the entries at the intersections of a set of rows $\{q'_{i_1}, ..., q'_{i_a}\}$ and a set of columns $\{q''_{j_1}, ..., q''_{j_b}\}$ are 1s then this set of 1s forms a* grid. *The grid is represented by $g = (q'_{i_1}, ..., q'_{i_a}; q''_{j_1}, ..., q''_{j_b})$. The grid g* contains *the pair (q'_i, q''_j) if $i \in \{i_1, ..., i_a\}$ and $j \in \{j_1, ..., j_b\}$. A set of*

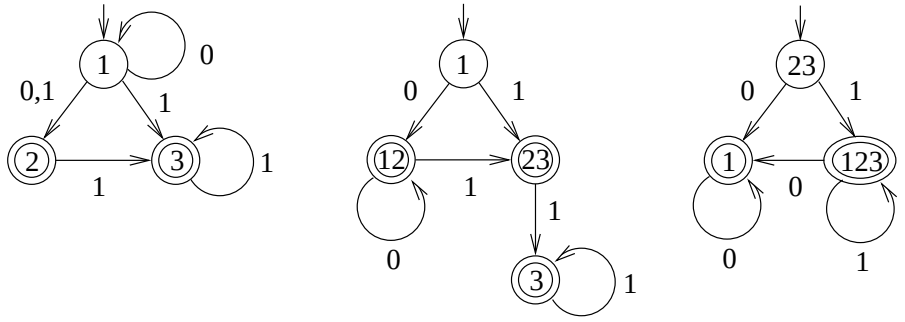

Figure 3.1: Automata A (left), $B = D(A)$ (center) and $C = D(A^R)$ (right)

grids forms a cover *if every 1 in the EAM (or RAM) belongs to at least one grid in the set. A* minimum cover *is a cover that consists of the minimum number of grids. Given a cover of an EAM (or RAM), the corresponding* cover map *is obtained by replacing each 1 in the EAM (or RAM) by the names of all the grids (in the given cover) it belongs to.*

Theorem 3.3 (Kameda and Weiner [24, Theorem 5]). *The SM (RSM) of an automaton A is a cover map, namely, the states of A appear as a cover of the EAM (RAM) of A.*

Example 3.1 *This example illustrates the concepts of this section. Figure 3.1 presents an automaton A and the corresponding automata $B = D(A)$ and $C = D(A^R)$. In Figure 3.2, the matrixes associated with the automaton A, namely, the SM, the EAM, the RSM and the RAM of A are shown. There is one minimum cover of the RAM of A consisting of two grids g_1 and g_2, such that $g_1 = (\{1\}, \{1,2\}; \{1\}, \{1,2,3\})$ and $g_2 = (\{1,2\}, \{2,3\}; \{1,2,3\}, \{2,3\})$. The cover map associated with the minimum cover is shown in Figure 3.3 (left).*

By a special rule, an NFA can be associated with any cover of the RAM of A. The rule is as follows. Let $\hat{B} = (\hat{Q}', \Sigma, \hat{\delta}', \hat{q}', \hat{F}')$, let Z be a cover of the RAM and let $f : \hat{Q}' \to 2^Z \setminus \{\emptyset\}$ be a function associated with Z which assigns to each state $\hat{p}$ of $\hat{B}$ the set of grids that intersect the row of the RAM that corresponds to $\hat{p}$. Then the NFA that is associated with the cover Z is given as $M = (Z, \Sigma, \gamma, Z_0, G)$ where for all $z \in Z$, $z' \in Z$, $\hat{p} \in \hat{Q}'$, and $a \in \Sigma$,

$Z_0 = f(\hat{q}')$,
$z \in G \Leftrightarrow (z \in f(\hat{p}) \Rightarrow \hat{p} \in \hat{F}')$, and
$z' \in \gamma(z, a) \Leftrightarrow (z \in f(\hat{p}) \Rightarrow z' \in f(\hat{\delta}'(\hat{p}, a)))$.

(a)

	{1}	{1,2,3}	{2,3}
{1}	{1}	{1}	
{1,2}	{1}	{1,2}	{2}
{2,3}		{2,3}	{2,3}
{3}		{3}	{3}

(b)

	{1}	{1,2,3}	{2,3}
{1}	1	1	
{1,2}	1	1	1
{2,3}		1	1
{3}		1	1

(c)

	{1}	{1,2,3}	{2,3}
{1}	{1}	{1}	
{1,2}	{1}	{1,2}	{2}
{2,3}		{2,3}	{2,3}

(d)

	{1}	{1,2,3}	{2,3}
{1}	1	1	
{1,2}	1	1	1
{2,3}		1	1

Figure 3.2: Matrixes associated with the automaton A: (a) the SM, (b) the EAM, (c) the RSM, (d) the RAM of A

	{1}	{1,2,3}	{2,3}
{1}	g_1	g_1	
{1,2}	g_1	g_1,g_2	g_2
{2,3}		g_2	g_2

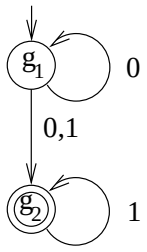

Figure 3.3: The cover map of the minimum cover $\{g_1, g_2\}$ of the RAM of A (left) and the NFA associated with that cover (right)

However, it may be the case that M is not equivalent to the original automaton A. To find a minimal NFA equivalent to A, [24] shows that an algorithm can be used that tests the covers of the RAM in increasing order of the size to find whether the NFA for the cover is equivalent to the original automaton. The first equivalent NFA found in this way is a minimal one. To check the equivalence of two automata, one may construct $D(M)$, find a minimal DFA equivalent to it and check if the resulting automaton is the same as $\hat{B}$, although [24] also proposes another procedure to accomplish the equivalence test.

Example 3.2 *Consider again the automaton A presented in Figure 3.1 and the minimum cover $\{g_1, g_2\}$ of the RAM of A as shown in Figure 3.3 (left). The NFA that is obtained from the cover $\{g_1, g_2\}$ by the special rule as discussed above is presented in Figure 3.3 (right). It can be easily verified that this automaton is equivalent to A, and therefore it is a minimal NFA equivalent to A.*

3.2 Bideterministic automata are minimal

An automaton is called bideterministic if both the automaton itself and its reversal automaton are deterministic. This means that a bideterministic automaton has a unique initial state and a unique final state, and there is at most one incoming and one outgoing transition with any label associated with any automaton state.

Bideterministic automata or bideterministic languages have been considered, for example, in the context of machine learning [2], as a special case of reversible automata and languages [29], and in coding theory [32]. It has been observed that a bideterministic automaton is a minimal DFA for the language [2, 29]. In coding theory bideterministic trellises — which are a very restricted class of bideterministic automata — are known to be minimal. This kind of trellises appear to correspond to certain codes (linear codes). It is well-known that a minimal deterministic trellis is a minimal trellis for such codes [28]. Here, the general case of bideterministic automata is considered.

It is known that a bideterministic automaton is minimal among the DFAs. This is very easy to show by using, for example, Brzozowski's minimization algorithm, which involves reversing, determinizing, again reversing and determinizing the automaton [4, 35]. As this algorithm, when applied to a bideterministic automaton, does not change it, it can be concluded that the automaton is the minimal DFA of its language. In the

following we show, using the theory in Section 3.1, that a bideterministic automaton is also minimal in the class of the NFAs.

Let $A = (Q, \Sigma, \delta, \{q_0\}, \{q_f\})$ be a bideterministic automaton. Its reversal automaton is $A^R = (Q, \Sigma, \delta^R, \{q_f\}, \{q_0\})$ where $\delta^R(p, a) = \{q\}$ if and only if $\delta(q, a) = \{p\}$ for all $p \in Q$, $q \in Q$, and $a \in \Sigma$. Then the automata B and C from Section 3.1 are simply $B = D(A) = A$ and $C = D(A^R) = A^R$.

Let $Q = \{q_0, ..., q_{N-1}\}$. According to Definition 3.1, the SM of A consists of N rows and N columns, with exactly N non-blank entries $\{q_0\}, ..., \{q_{N-1}\}$ in the matrix which are positioned so that there is exactly one such entry in every row and every column. The corresponding EAM is basically the same as SM, only these non-blank entries are replaced with 1s. As there are no two equivalent rows nor columns in SM, it follows from Definition 3.2 that the RSM and RAM of A are the same as SM and EAM, respectively. Since there is exactly one 1 in every row and in every column of the RAM of A, we see by Definition 3.3 that every grid in the RAM contains just one row-column pair. Altogether there are N such grids in the RAM, and moreover, this set of grids is the only cover of the RAM. Because the RAM is unique for all automata accepting $L(A)$ (Theorem 3.2) and any automaton accepting $L(A)$ has to have at least as many states as is the number of grids in the minimum cover of RAM (Theorem 3.3), we conclude that A is a minimal automaton. We have proved the following theorem:

Theorem 3.4 *A bideterministic automaton is minimal among all finite automata accepting the same language.*

Next, we show that a bideterministic automaton is uniquely minimal, that is, there does not exist any other automaton with the same number of states that accepts the same language. For this, we first note that, as we discussed above, a bideterministic automaton is a minimal DFA which is known to be unique. Therefore, if any other automaton with the same number of states exists, it has to be non-deterministic. But this kind of automata do not exist, as the following lemma shows.

Lemma 3.2 *Any non-deterministic automaton equivalent to a bideterministic automaton A has more states than A does.*

Proof. Let A be a bideterministic automaton with N states. Consider any non-deterministic automaton A' that accepts the same language as A does. This means that either A' has multiple initial states or there is a state in A' from which there are transitions with the same label to more than one state. In any case, the determinized automaton $D(A')$ must have a state p

— a subset of states of A' — of cardinality more than one. Let $p_1, ..., p_m$ be the states of A' comprising that subset, $m > 1$. Now, the row $\hat{p}$ in the states map SM of A' corresponding to state p has to be such that every $p_j, j = 1, ..., m$, belongs to at least one entry in that row (because every state of A' belongs to some state of $D((A')^R)$). But, as we know that the RAM of A' is the same as the RAM of A (Theorem 3.2), then, according to the properties of the RAM of a bideterministic automaton as shown above, the RAM of A' has N rows and N columns with exactly one 1 in each row and each column. Hence the RSM of A' has exactly one non-blank entry in each row and each column. As an RSM is formed by merging the equivalent rows and columns of a SM (Definition 3.2), the RSM of A' must have a row — namely the row that contains the row $\hat{p}$ of the SM of A' — whose only non-blank entry contains all $p_j, j = 1, ..., m$ (and possibly some other states of A'). It also has to be the case that the intersection of any two entries of the RSM of A' is empty, or otherwise the RSM of A' could not have just one non-blank entry in each row and column. Because there are N rows and N columns and thus N non-blank entries in the RSM of A', it follows that A' has at least $N - 1 + m$ states. As we had $m > 1$, we get that A' must have more than N states. □

As a conclusion we may state the following theorem:

Theorem 3.5 *A bideterministic automaton is uniquely minimal.*

Proof. Follows from Lemma 3.2 and from the fact that a bideterministic automaton is the minimal DFA which is unique. □

Remark 3.1 *The proof of Theorem 3.5 is independent from the result of Theorem 3.4. So, Theorem 3.4 follows from Theorem 3.5.*

Example 3.3 *Let $L_k = \{ww^R | w \in \{0,1\}^k\}$ where $k \geq 0$, be a set of strings consisting of concatenations of any binary string of length k and its reversal string. Let L_k^* be the set that consists of strings obtained by concatenating zero or more times the elements of L_k. Then for every $k \geq 0$, L_k^* is accepted by a bideterministic automaton having $3 \times 2^k - 3$ states; the leftmost automaton in Figure 3.4 is such an automaton with 9 states accepting L_2^*. By Theorem 3.4 we know that this is a minimal automaton recognizing this language and we cannot find a smaller automaton representation for it.*

Example 3.4 *The language $L((\mathbf{0}+\mathbf{1})((\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1}))^*)$ consisting of all odd-length binary strings is accepted by the bideterministic automaton shown*

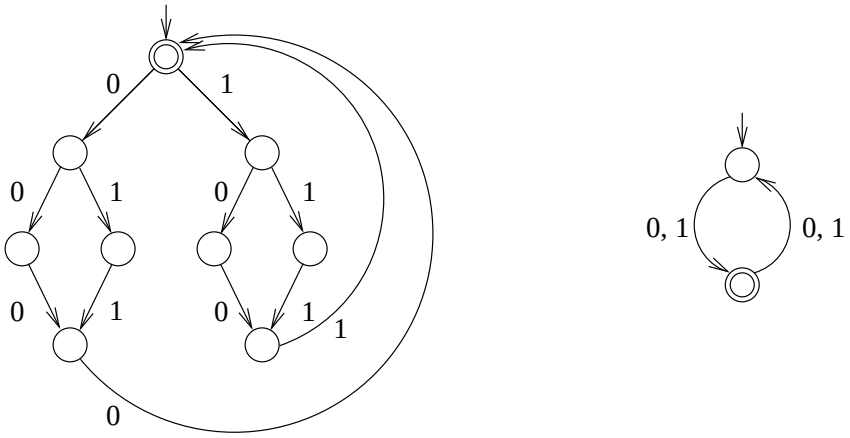

Figure 3.4: Minimal automaton of the language L_2^* of Example 3.3 (left) and minimal automaton of the language $L((\mathbf{0}+\mathbf{1})((\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1}))^*)$ (right)

as the rightmost automaton in Figure 3.4. By Theorem 3.4, this is a minimal automaton accepting this language.

We showed above that, given a bideterministic automaton A, any other automaton equivalent to A has more states than A does. Let A' be an automaton, different from A, with $L(A') = L(A)$. Then A' has more states than A. In the following we show that A has at most as many transitions as A' does. Consider the RSM of A'. It must have exactly one nonblank entry in every row and column because the RAM of A (and A') has this property. Also, it must be the case that the intersection of any two entries of the RSM of A' is empty. The non-blank entries of the RSM correspond to the states of the automaton $\hat{B}'$ that is obtained by merging the equivalent states of $B' = D(A')$. As the sets corresponding to any two states of $\hat{B}'$ do not intersect, then any state of A' appears only in one state of $\hat{B}'$. As $\hat{B}'$ is deterministic then the number of transitions leaving any state of $\hat{B}'$ is at most the sum of the numbers of outgoing transitions of the states of A' belonging to this state. Thus, the overall number of transitions in $\hat{B}'$ is at most the same as the number of transitions in A'. As $\hat{B}' = A$, we have obtained the following result:

Theorem 3.6 *A bideterministic automaton has the minimum number of transitions.*

3.3 More minimality results

From Theorem 3.4 we know that bideterminism is a sufficient condition for a language to have the property that the size of its minimal deterministic automaton is also the smallest possible size of any (nondeterministic) automaton accepting the same language. It is of interest to find other conditions that imply similar minimalities. In this section, we present another, more general minimality result. Actually, Theorem 3.4 follows from this result as a special case.

The discussion below is based on two automata defined as follows. First, let $A = (Q, \Sigma, \delta, \{q\}, F)$ be a minimal deterministic automaton, and second, let $A_1 = D(A^R) = (Q'', \Sigma, \delta'', \{q''\}, F'')$ be the automaton obtained from the reversal of A by the subset construction. Every state q''_i of A_1 is a subset of the state set Q of A.

Consider the partitions of the state sets of A and A_1, defined in the following way. Let $\{Q''_1, ..., Q''_k\}$ be the partition of the state set Q'' of A_1 into disjoint subsets (equivalence classes) such that any pair of states q''_1 and q''_2 of A_1 belongs to the same Q''_i, $i \in \{1, ..., k\}$, if and only if there exist states $q''_{i_1}, ..., q''_{i_l}$ of A_1 such that $q''_{i_1} = q''_1$, $q''_{i_l} = q''_2$ and $q''_{i_j} \cap q''_{i_{j+1}} \neq \emptyset$ for all $j = 1, ..., l-1$. And let $\{Q_1, ..., Q_k\}$ be the partition of the state set Q of A into disjoint subsets such that $Q_i = \bigcup_{q''_j \in Q''_i} q''_j$ for $i = 1, ..., k$. More intuitively, we divide the state set Q'' of A_1 into disjoint subsets in a way which ensures that the states of A_1 (as subsets of Q) belonging to different subsets of the partition do not intersect. Such partition of the states of A_1 accordingly induces a partition of the state set Q of A as well if we divide Q into subsets so that each subset is a union of the states belonging to the corresponding partition subset of Q''. These partitions divide the RAM of A into disjoint submatrixes, and in order to find a minimum cover of the RAM one can find a minimum cover for each such submatrix and take the union of those covers as it will be shown shortly below.

Consider the theory of Section 3.1 in the case where the automaton A of that section is the automaton A given above. Then the automata B and C of Section 3.1 are equal to the automata A and A_1, respectively, as $B = D(A) = A$ and $C = D(A^R) = A_1$. As B is equal to A, B is the minimal DFA accepting $L(A)$. According to the Brzozowski's minimization algorithm, C is the minimal DFA accepting $L(A^R)$ as we can write $C = D(A^R) = D(D(A)^R) = D(D((A^R)^R)^R)$. Hence the SM of A is also the RSM of A (Definitions 3.1 and 3.2, Lemma 3.1). The number of rows and columns in the RSM (and the RAM) of A equals the number of states of A and A_1, respectively. There is a one-to-one correspondence between the

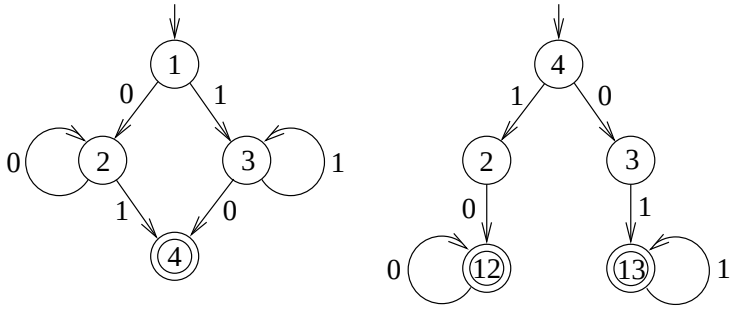

Figure 3.5: Minimal DFA A of the language $L(\mathbf{00^*1 + 11^*0})$ and $A_1 = D(A^R)$

states of A and the rows of the RAM of A, as well as between the states of A_1 and the columns of the RAM of A. We denote by RAM_i, where $i \in \{1, ..., k\}$, the submatrix of the RAM of A, which is formed from the rows corresponding to the states belonging to Q_i and from the columns corresponding to the states in Q_i''.

Similarly to Definition 3.3, we say that a set of grids covers some RAM_i if every 1 in that RAM_i belongs to at least one grid in the set. Also, we say that a set of grids covers a set of rows and columns of the RAM if every 1 in these rows and columns belongs to at least some grid in that grid set.

Before we start to develop a theory based on the concepts introduced above, we present the following example to illustrate these concepts.

Example 3.5 *In Figure 3.5, the leftmost automaton is the minimal DFA A accepting the language $L(\mathbf{00^*1 + 11^*0})$ and the rightmost automaton $A_1 = D(A^R)$ is obtained from the reversal of A by the subset construction. The state set Q of A is the set $\{1, 2, 3, 4\}$, and the state set Q'' of A_1 is the set $\{\{1, 2\}, \{1, 3\}, \{2\}, \{3\}, \{4\}\}$. The partitions of Q and Q'' divide both sets into two subsets: $Q_1 = \{1, 2, 3\}$, $Q_2 = \{4\}$, $Q_1'' = \{\{1, 2\}, \{1, 3\}, \{2\}, \{3\}\}$, $Q_2'' = \{\{4\}\}$. The RAM of A is shown in Figure 3.6.*

Lemma 3.3 *Let G_i, where $i = 1, ..., k$, be any minimal set of grids covering the RAM_i. Then $G_i \cap G_j = \emptyset$ for $i \neq j$, $i, j = 1, ..., k$, and $G_1 \cup ... \cup G_k$ is a minimum cover of the RAM of A.*

Proof. It is clear that any set of grids covers the RAM_i if and only if it covers the set of rows and columns corresponding to Q_i and Q_i''. According to the definition of the partition of Q'', any two states taken from Q_i'' and

	{1,2}	{1,3}	{2}	{3}	{4}
{1}	1	1			
{2}	1		1		
{3}		1		1	
{4}					1

Figure 3.6: The RAM of A

Q''_j, where $i \neq j$, have an empty intersection. Therefore, any two 1s in the RAM of A, where the first 1 is in a column that corresponds to some state of Q''_i and the second 1 is in a column corresponding to some state of Q''_j, cannot belong to the same grid. It follows that $G_i \cap G_j = \emptyset$. In order to find a minimum cover of the RAM, one can find a minimal set of grids covering the columns of Q''_i (this set automatically also covers the rows of Q_i) for every $i \in \{1, ..., k\}$, and take the union of these sets. □

Definition 3.4 *A grid is* elementary *if it consists of just one 1, that is, one row and one column. A grid with two or more 1s in it, that is, two or more rows or columns, is* non-elementary*. A non-elementary grid is* horizontal *(*vertical*) if all of its 1s are in the same row (column), that is, if it consists of one row (column).*

Lemma 3.4 *Consider the following three conditions:*
(a) Every state of A_1 consists of at most two states of A.
(b) Each state of A occurs in at most two states of A_1.
(c) Any two states of A_1 have at most one state of A in common.
If any of the conditions (a), (b) and (c) holds then every non-elementary grid in the RAM of A is either horizontal or vertical.

Proof. Suppose that there is a non-elementary grid in the RAM of A that is not horizontal nor vertical. This means that there is a grid $g = (q_{i_1}, ..., q_{i_a}; q''_{j_1}, ..., q''_{j_b})$ such that $a \geq 2$ and $b \geq 2$. This implies that there are at least two states q''_{j_1} and q''_{j_2} of A_1, both of which contain the states q_{i_1} and q_{i_2} of A. If (a) holds then the columns of the RAM of A corresponding to q''_{j_1} and q''_{j_2} must be equivalent. This is not possible, thus we have a contradiction. If (b) holds then the rows corresponding to q_{i_1} and q_{i_2} must be equivalent. This is not possible either, thus we

have a contradiction in this case, too. If (c) holds then we have a direct contradiction. □

Lemma 3.5 *If a RAM_i, $i \in \{1, ..., k\}$, has more than one row or column and if every non-elementary grid in that RAM_i is either horizontal or vertical then there is a minimal set of grids covering the RAM_i, consisting of only horizontal and vertical grids, such that every horizontal grid in that set covers the corresponding row and every vertical grid in that set covers the corresponding column.*

Proof. Let the assumption of the lemma hold and let G_i be any minimal set of grids covering the RAM_i. We modify G_i as follows. First, we observe that if G_i contains an elementary grid then there has to be two or more 1s in the row or the column involved by such a grid, otherwise the RAM_i would be a 1×1 matrix. Therefore we can and do replace any elementary grid in G_i either with a horizontal grid covering the corresponding row or with a vertical grid covering the corresponding column. Also, we replace any horizontal grid in G_i, which does not cover its corresponding row, with the horizontal grid covering that row entirely, and we replace any vertical grid in G_i, which does not cover its corresponding column, with the vertical grid covering that column entirely. After these replacements G_i still covers the RAM_i and since the number of grids in G_i does not increase in the process (neither can it decrease because G_i was minimal in the beginning), G_i stays minimal. Thus, the modified G_i is as claimed in the lemma. □

Before we present our main lemma of this section — Lemma 3.7 —, we present a simple Lemma 3.6 that is used by the main lemma. The result of Lemma 3.6 is presented with proof in [34].

Lemma 3.6 (Tamm and Ukkonen [34, Lemma 24(c)]). *Let K be a collection of non-empty subsets of $\{1, 2, \ldots, n\}$. If any two different members of K have at most one element in common, then $|K| \leq n(n+1)/2$.*

Lemma 3.7 *Consider Q_i and Q_i'', $i \in \{1, ..., k\}$, and assume that at least one of the following three conditions holds:*
(a) Every state of A_1 consists of at most two states of A.
(b) Each state of A occurs in at most two states of A_1.
(c+) Any two states of A_1 have at most one state of A in common, and one of the next three conditions is true: (i) $|Q_i| \leq 4$ or $|Q_i''| \leq 4$, (ii) $|Q_i| \geq 5$ and $|Q_i''| > |Q_i|(|Q_i|-5)/2+5$, (iii) $|Q_i''| \geq 5$ and $|Q_i| > |Q_i''|(|Q_i''|-5)/2+5$. Then the minimum number of grids that cover the RAM_i is the minimum of $|Q_i|$ and $|Q_i''|$.

Proof. Let at least one of the above conditions (a), (b), (c+) hold. If the RAM_i is a 1×1 matrix then it is clear that the statement of the lemma holds. Now consider the case where the RAM_i has more than one row or column. According to Lemma 3.4, any grid in the RAM_i is either horizontal, vertical or an elementary grid. Let G_i be a minimal set of grids covering RAM_i, which has the properties specified in Lemma 3.5. Then we may see G_i as the union of two non-intersecting sets of grids, denoted by G_i^h and G_i^v where the set G_i^h consists of horizontal grids where every grid covers its corresponding row and G_i^v consists of vertical grids with every grid covering the corresponding column. One of these two sets may be empty.

Obviously, the number of grids in G_i cannot be larger than the minimum of $|Q_i|$ and $|Q_i''|$. Neither can it be smaller, as we show in the following by contradiction.

Suppose that $|G_i| < |Q_i|$ and $|G_i| < |Q_i''|$. This implies that both G_i^h and G_i^v must be non-empty. As $|G_i^h| + |G_i^v| < |Q_i|$, then from the fact that G_i^h covers $|G_i^h|$ rows it follows that G_i^v must cover the remaining of the $|Q_i|$ rows, i.e., at least $|Q_i| - |G_i^h| > |G_i^v|$ rows. Also, from $|G_i^h| + |G_i^v| < |Q_i''|$ and from the fact that G_i^v covers $|G_i^v|$ columns it follows that G_i^h must cover at least $|Q_i''| - |G_i^v| > |G_i^h|$ columns. Due to the fact that the RAM_i cannot have equivalent columns nor rows, one horizontal grid can cover at most one column and one vertical grid can cover at most one row. Therefore, both G_i^h and G_i^v have to consist of at least two grids. Thus $|G_i^h| \geq 2$ and $|G_i^v| \geq 2$ which implies $|G_i| \geq 4$.

If (a) holds then there are exactly two 1s in every grid of G_i^v. This way, the grids in G_i^v can involve at most $|G_i^v| + 1$ rows. We had above that G_i^v must cover more than $|G_i^v|$ rows thus we conclude that the grids in G_i^v cover all the rows that they involve. But this implies $G_i^v = G_i$, so we have obtained a contradiction.

If (b) holds then there are exactly two 1s in every grid of G_i^h. This way, the grids in G_i^h can involve at most $|G_i^h| + 1$ columns. As we had above that G_i^h must cover more than $|G_i^h|$ columns then the grids in G_i^h cover all the columns that they involve. But this implies $G_i^h = G_i$, a contradiction.

If (c+) holds then in case (i) $|Q_i| \leq 4$ or $|Q_i''| \leq 4$ the hypothesis that $|G_i| < |Q_i|$ and $|G_i| < |Q_i''|$ leads to a contradiction since we showed above that $|G_i| \geq 4$. Now consider the case (ii) $|Q_i| \geq 5$ and $|Q_i''| > |Q_i|(|Q_i| - 5)/2 + 5$. Then we have $|Q_i''| > |Q_i|$ which means that $|G_i| \leq |Q_i| - 1$. Therefore $|G_i^h| + |G_i^v| \leq |Q_i| - 1$, which can also be written as $|Q_i| \geq |G_i^h| + |G_i^v| + 1$. Using this last inequality with $|Q_i''| > |Q_i|(|Q_i| - 5)/2 + 5$, we get that $|Q_i''| > (|G_i^h| + |G_i^v| + 1)(|G_i^h| + |G_i^v| - 4)/2 + 5$. By Lemma 3.6 we

know that $|G_i^h|$ horizontal grids can cover at most $|G_i^h|(|G_i^h|+1)/2$ columns. Vertical grids in G_i^v cover $|G_i^v|$ columns. Therefore $|Q_i''| \leq |G_i^h|(|G_i^h| + 1)/2 + |G_i^v|$. Putting two last inequalities together, we get $|G_i^h|(|G_i^h| + 1)/2 + |G_i^v| > (|G_i^h| + |G_i^v| + 1)(|G_i^h| + |G_i^v| - 4)/2 + 5$ from which it follows $(|G_i^v| - 2)(2|G_i^h| + |G_i^v| - 3) < 0$. But this cannot hold, since neither $|G_i^v| - 2$ nor $2|G_i^h| + |G_i^v| - 3$ can be negative. Thus we have obtained a contradiction. The proof for the case (iii) $|Q_i''| \geq 5$ and $|Q_i| > |Q_i''|(|Q_i''| - 5)/2 + 5$ is symmetric and similar to the case (ii). □

The main result of this section is in the form of the following theorem:

Theorem 3.7 *Consider a minimal DFA A and $A_1 = D(A^R)$ with the partitions of their state sets $\{Q_1, ..., Q_k\}$ and $\{Q_1'', ..., Q_k''\}$ as described above. Assume that at least one of the following three conditions holds:*
(a) Every state of A_1 consists of at most two states of A.
(b) Each state of A occurs in at most two states of A_1.
(c+) Any two states of A_1 have at most one state of A in common, and for every $i = 1, ..., k$ one of the next three conditions is true: (i) $|Q_i| \leq 4$ or $|Q_i''| \leq 4$, (ii) $|Q_i| \geq 5$ and $|Q_i''| > |Q_i|(|Q_i| - 5)/2 + 5$, (iii) $|Q_i''| \geq 5$ and $|Q_i| > |Q_i''|(|Q_i''| - 5)/2 + 5$.

If $|Q_i| \leq |Q_i''|$ for all $i = 1, ..., k$, then A is a minimal automaton accepting $L(A)$. If $|Q_i''| \leq |Q_i|$ for all $i = 1, ..., k$, then A_1^R is a minimal automaton accepting $L(A)$.

Proof. According to the assumptions of this theorem and Lemma 3.7, the minimum number of grids to cover the RAM_i, for $i = 1, ..., k$, is the minimum of $|Q_i|$ and $|Q_i''|$. By Lemma 3.3, if $|Q_i| \leq |Q_i''|$ for all $i = 1, ..., k$, then a minimum cover of the RAM of A consists of $|Q_1| + ... + |Q_k| = |Q|$ grids. Similarly, if $|Q_i''| \leq |Q_i|$ for all $i = 1, ..., k$, then a minimum cover of the RAM of A consists of $|Q_1''| + ... + |Q_k''| = |Q''|$ grids. We know that any automaton equivalent to A cannot have less states than is the number of grids in a minimum cover of the RAM of A (Theorems 3.2 and 3.3). We know that A and A_1^R both accept $L(A)$ and their sizes are $|Q|$ and $|Q''|$, respectively. Therefore, if $|Q_i| \leq |Q_i''|$ for all $i = 1, ..., k$, then A is a minimal automaton accepting $L(A)$. Also, if $|Q_i''| \leq |Q_i|$ for all $i = 1, ..., k$, then A_1^R is a minimal automaton accepting $L(A)$. □

Example 3.6 *Consider the automata A and A_1, presented in Figure 3.5. The partitions of the state sets of A and A_1 were shown in Example 3.5. It can be easily verified that for these automata, all three conditions (a), (b) and (c+) of Theorem 3.7 hold. As $|Q_1| = 3 < 4 = |Q_1''|$ and $|Q_2| = 1 = |Q_2''|$ then by Theorem 3.7, A is a minimal automaton.*

Corollary 3.1 *Theorem 3.4 follows from Theorem 3.7.*

Proof. Let A be a bideterministic automaton. Then A is a minimal DFA. The automaton A_1 of Theorem 3.7 is $A_1 = A^R$. Since the state sets of A and A_1 coincide and $|Q_i| = |Q_i''| = 1$ for all $i = 1, ..., k$, it is clear that all three conditions (a), (b) and (c+) of Theorem 3.7 hold. As both $|Q_i| \leq |Q_i''|$ and $|Q_i''| \leq |Q_i|$ are true for all $i = 1, ..., k$, then we may conclude that both A and $(A^R)^R$ are minimal automata accepting $L(A)$. But these two automata are the same, so Theorem 3.4 follows. □

In certain cases Theorem 3.7 gives two different minimal automata:

Corollary 3.2 *Let A and A_1 be automata meeting the assumptions of Theorem 3.7. If $|Q_i| = |Q_i''|$ for all $i = 1, ..., k$, and A is not bideterministic then A and A_1^R are two different minimal automata accepting $L(A)$.*

Proof. As $|Q_i| \leq |Q_i''|$ and $|Q_i''| \leq |Q_i|$ for all $i = 1, ..., k$, then, according to Theorem 3.7, both A and A_1^R are minimal. If A is not bideterministic then A_1^R has to be nondeterministic. So, A and A_1^R must be different. □

Example 3.7 *Figure 3.7 presents two automata given by [3] as examples of two different minimal automata accepting the language $\{ab, ac, ba, bc, ca, cb\}$. In terms of our theory, the leftmost automaton is the minimal DFA A accepting that language and the rightmost one is $A_1^R = (D(A^R))^R$. The partitions of the state sets of A and A_1 are the following: $Q_1 = \{1\}$, $Q_2 = \{2, 3, 4\}$, $Q_3 = \{5\}$, $Q_1'' = \{\{1\}\}$, $Q_2'' = \{\{2, 3\}, \{2, 4\}, \{3, 4\}\}$, $Q_3'' = \{\{5\}\}$. As the assumptions of Theorem 3.7 hold, $|Q_i| = |Q_i''|$ for all $i = 1, 2, 3$, and A is not bideterministic then by Corollary 3.2 both A and A_1^R are minimal, indeed.*

The sufficiency conditions of Theorem 3.7 can be checked in polynomial time. These matters are discussed in [34] which presents the following result:

Theorem 3.8 (Tamm and Ukkonen [34, Theorem 25]). *For a minimal DFA A with n states, one can test in time $O(n^4 \log n)$ whether or not A satisfies the sufficiency conditions of Theorem 3.7 for minimality.*

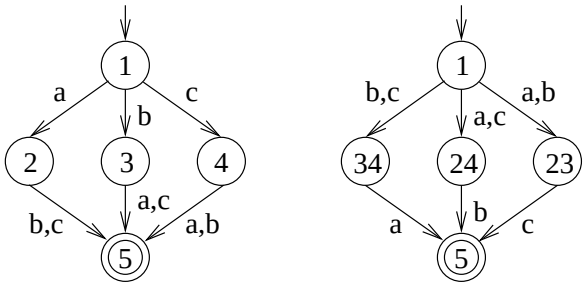

Figure 3.7: Minimal DFA A of the language $\{ab, ac, ba, bc, ca, cb\}$ and $A_1^R = (D(A^R))^R$

Chapter 4

Bideterministic multitape automata

In Chapter 3 we proved that a (one-tape) bideterministic automaton is minimal. Here we consider the Rabin-Scott one-way multitape automaton model and show that the minimality property does not hold for bideterministic multitape automata. Still, a bideterministic multitape automaton is a unique minimal automaton among the automata which have the same set of accepting computations.

4.1 Reversal of a multitape automaton

In this section we define the notion of a reversal of an automaton in our given multitape model.

Let A be a one-way n-tape automaton $(Q, tape, \Sigma, \delta, I, F)$ with a partition of Q into $Q_1, ..., Q_n$ in the Rabin-Scott model. That is, every state is associated with a tape to be read when the control is in that state. To get a reversal automaton A^R of the automaton A, it is not enough if we just reverse the transition function and swap the sets of initial and final states of A, as in the definition of a reversal for a one-tape automaton in Section 2.1. In addition to these, we possibly also have to change the state-tape associations in A^R because given a transition in A leading from a state q_1 to a state q_2 it is possible that the tape t_1 assigned to q_1 differs from the tape t_2 assigned to q_2. In this case we have to associate (in A^R) q_2 with the tape t_1 instead. Furthermore, it is possible that there are transitions in A both from q_1 and q_2 to a state q_3 while the tapes associated with q_1 and q_2 are not the same. If this is the case then we consider the reversal of A to be undefined. Otherwise, one more thing to be done is to change

the labels of the transitions in A^R that involve endmarkers; specifically, all transitions labelled with the left endmarker have to be labelled with the right endmarker and vice versa.

Now, more formally, if there are states $q_1 \in Q$ and $q_2 \in Q$ such that $tape(q_1) \neq tape(q_2)$ and $\delta(q_1, a) \cap \delta(q_2, b) \neq \emptyset$ for some $a, b \in \Sigma \cup \{[,]\}$ then the reversal of A is undefined. Otherwise, let us define a function $tape^R : Q \to \{1, ..., n\}$ as follows. Let $q \in Q$. If there is some $q_1 \in Q$ such that $\delta(q_1, a) = q$ for some $a \in \Sigma \cup \{[,]\}$ then $tape^R(q) = tape(q_1)$, otherwise let $tape^R(q) = tape(q)$.

The reversal of A is the automaton A^R given by $(Q, tape^R, \Sigma, \delta^R, F, I)$ with a partition of Q into $Q_1^R, ..., Q_n^R$ so that $Q_i^R = \{q \in Q \mid tape^R(q) = i\}$ for $i = 1, ..., n$, where for all $q_1 \in Q$ and $a \in \Sigma$, $\delta^R(q_1, a) = \{q \mid q_1 \in \delta(q, a)\}$, $\delta^R(q_1, [) = \{q \mid q_1 \in \delta(q,])\}$ and $\delta^R(q_1,]) = \{q \mid q_1 \in \delta(q, [)\}$.

4.2 Bideterministic multitape automata

In Chapter 3 we discussed bideterministic one-tape automata. In this section we extend the notion of bideterminism to one-way multitape automata and show two simple properties of bideterministic multitape automata.

We call a multitape automaton A *bideterministic* if its reversal A^R is defined and both A and A^R are deterministic.

In Section 3.2 it was proved that a one-tape bideterministic automaton is a minimal automaton accepting its language. But for multitape automata this result does not necessarily hold in the general case. We show this by a counterexample. In Figure 4.1 two equivalent bideterministic multitape automata accepting the language $\{(ab, a), (bc, a)\}$ are presented. The labels shown at the transitions are indexed labels, that is, an index of a symbol indicates the tape which is read. As the leftmost automaton has more states than the other one then it clearly cannot be a minimal automaton. Therefore in the multitape case bideterminism is not a sufficient condition for minimality. The following statement holds:

Proposition 4.1 *A bideterministic multitape automaton is not necessarily minimal.*

The second property of a bideterministic multitape automaton which we present here is as follows:

Proposition 4.2 *A bideterministic multitape automaton A is the unique minimal automaton with accepting computations $C(A)$.*

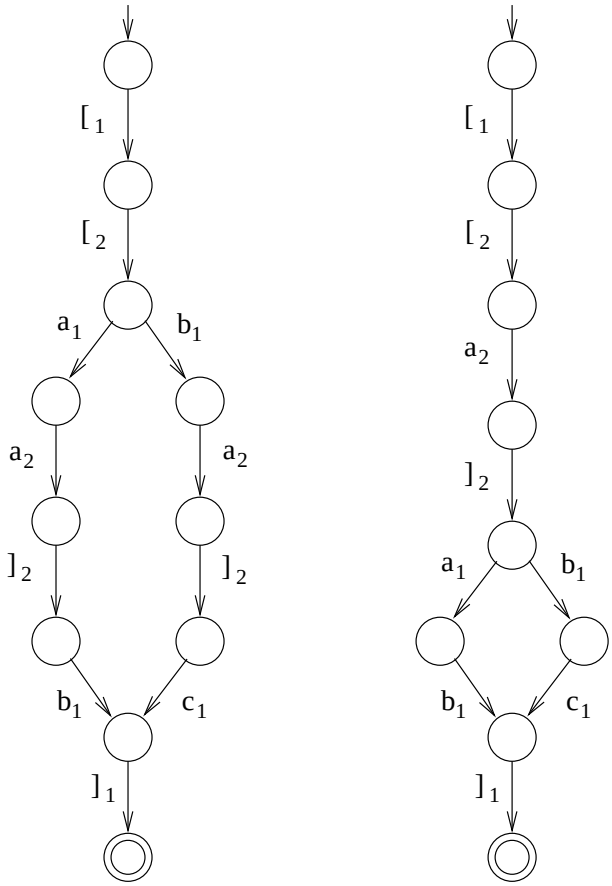

Figure 4.1: Two different bideterministic two-tape automata accepting the language $\{(ab, a), (bc, a)\}$ showing that the leftmost automaton is not minimal. The indexes of the labels at the transitions indicate the tapes that are read.

Proof. Let A be a bideterministic multitape automaton. Let us consider the one-tape automaton A' that is obtained from A by replacing the transition labels of A with their corresponding indexed labels and discarding the state-tape associations. As an accepting computation of a one-tape automaton is just a string obtained by concatenating the transition labels on an accepting path, then $C(A') = C(A)$. Because A' has only one tape, it holds that $L(A') = C(A')$. Also, A' is bideterministic.

Suppose now that there is another multitape automaton A_1 different from A with $C(A_1) = C(A)$ so that the size of A_1 is less than or equal to the size of A. Let A'_1 be its one-tape counterpart obtained from A_1 as described above. Then again $L(A'_1) = C(A'_1) = C(A_1)$. But we have that $C(A_1) = C(A) = C(A') = L(A')$ and so the equality $L(A'_1) = L(A')$ holds. As A'_1 is different from A' and $|A'_1| = |A_1| \leq |A| = |A'|$ then A' cannot be the unique minimal automaton accepting its language. But this is a contradiction to what we have shown in Section 3.2 that a bideterministic one-tape automaton is uniquely minimal.

We may conclude that A is the unique minimal automaton having the set of accepting computations $C(A)$. □

Chapter 5

Size reduction of multitape automata

In this chapter we consider a size reduction algorithm for multitape automata. It is assumed that an automaton is in the mixed-state model and does not have ϵ-transitions. This algorithm tries to reduce the automaton size by changing the order in which some transitions concerning different tapes are performed, and combining some suitable states into a single state. First, in Section 5.1 we describe the four main operations we use to transform an automaton, and then in Section 5.2 we present an algorithm that uses these operations to reduce the size of the automaton. In Section 5.3 we analyze the time and space requirements of the algorithm. Finally we present an example of a family of automata for which the algorithm works nicely.

5.1 Simple automata transformations

In this section we describe four simple transformations on multitape automata that we find useful in our reduction algorithm. These transformations will be seen to be correct in the sense that they do not change the language an automaton accepts.

Let A be an n-tape automaton $(Q, \Sigma, \delta, I, F)$. We start with the following observation:

Observation 5.1 *If there are transitions* $q_1 \xrightarrow{a_i} q_2$ *and* $q_2 \xrightarrow{b_j} q_3$ *in* A, *where* $q_1, q_2, q_3 \in Q$, $a, b \in \Sigma$ *and* $i, j \in \{1, ..., n\}$, *such that* $i \neq j$ *and* q_2 *is not an initial nor an accepting state and has exactly one incoming and one outgoing transition, then the labels of these two transitions can be interchanged without changing the language of* A.

To see the correctness of Observation 5.1, we note that any accepting path in A either takes both transitions $q_1 \xrightarrow{a_i} q_2$ and $q_2 \xrightarrow{b_j} q_3$ or takes none of them. This is because these two transitions are the only ones involving q_2, and q_2 is not an initial nor an accepting state. Also, these transitions involve different automata tapes i and j reading of which is independent from one another. Based on this, it is clear that if in q_1, instead of these transitions, an accepting path would take a transition $q_1 \xrightarrow{b_j} q_2$ followed by $q_2 \xrightarrow{a_i} q_3$ then the contents of accepted strings would not change.

The transformation in Observation 5.1 has a limited applicability because of the restrictions it has. If we drop the requirement that q_2 has only one incoming transition but instead allow several transitions each of which involves some tape other than j to enter q_2 then we can have another, more general transformation which we call *Swap Upwards*, described as follows.

Swap Upwards. Let $q, q', q_1, ..., q_k \in Q$. Let there be transitions $q_1 \xrightarrow{(a_1)_{i_1}} q', ..., q_k \xrightarrow{(a_k)_{i_k}} q'$ and $q' \xrightarrow{b_j} q$ in A, such that j refers to a tape that is different from all tapes $i_l, l \in \{1, ..., k\}$. Let q' be a non-initial and non-final state with $outdegree(q') = 1$ and $indegree(q') = k$. Then q' and its incoming and outgoing transitions can be removed and replaced with new non-initial and non-final states $q'_1, ..., q'_k$ and transitions $q_1 \xrightarrow{b_j} q'_1, ..., q_k \xrightarrow{b_j} q'_k$, and $q'_1 \xrightarrow{(a_1)_{i_1}} q, ..., q'_k \xrightarrow{(a_k)_{i_k}} q$.

To show the correctness of the Swap Upwards transformation, we can consider this transformation as a sequence of the following transformations. First, q' with its incoming and outgoing transitions is removed and replaced with $q'_1, ..., q'_k$ and transitions $q_1 \xrightarrow{(a_1)_{i_1}} q'_1, ..., q_k \xrightarrow{(a_k)_{i_k}} q'_k$, and $q'_1 \xrightarrow{b_j} q, ..., q'_k \xrightarrow{b_j} q$. Second, for all $l = 1, ..., k$, the transformation described in Observation 5.1 is applied for transitions $q_l \xrightarrow{(a_l)_{i_l}} q'_l$ and $q'_l \xrightarrow{b_j} q$. The correctness of the first transformation is obvious if we consider its reverse transformation: it is clear that the states $q'_1, ..., q'_k$ (as after the first transformation) are equivalent and can be replaced by a new single state thus giving us the original automaton back again. The correctness of the other transformations is clear from the correctness of Observation 5.1.

Intuitively, the Swap Upwards transformation "moves" a transition upwards in the automaton graph. Next we define a similar transformation called *Swap Downwards* which acts in the opposite direction.

Swap Downwards. Let $q, q', q_1, ..., q_k \in Q$. Let there be transitions $q \xrightarrow{b_j} q'$, $q' \xrightarrow{(a_1)_{i_1}} q_1, ..., q' \xrightarrow{(a_k)_{i_k}} q_k$ in A such that $j \neq i_l$ where $l \in \{1, ..., k\}$. Let q' be a non-initial and non-final state with $indegree(q') = 1$ and $outdegree(q') = k$. Then q' and its incoming and outgoing transitions can be removed and replaced with new non-initial and non-final states $q'_1, ..., q'_k$ and transitions $q \xrightarrow{(a_1)_{i_1}} q'_1, ..., q \xrightarrow{(a_k)_{i_k}} q'_k$, and $q'_1 \xrightarrow{b_j} q_1, ..., q'_k \xrightarrow{b_j} q_k$.

The correctness of the Swap Downwards transformation can be shown similarly to the Swap Upwards operation.

The transformations Swap Upwards and Swap Downwards can be used to "move" transitions in the automaton graph, thus making it possible to "combine" some states of the automaton into one state, as described by the following two transformations.

Sink Combine. Let $q_1, ..., q_k$ be some non-initial states of A, all having exactly one incoming transition with the same label a_i from a state q of A where q is different from all q_l, $l \in \{1, ..., k\}$. Then $q_1, ..., q_k$ can be combined into one state q', meaning that $q_1, ..., q_k$ and their incoming and outgoing transitions are removed and replaced by a new non-initial state q' which is a final state if and only if any of $q_1, ..., q_k$ is final, with all outgoing transitions of $q_1, ..., q_k$ now leaving q', and the transition $q \xrightarrow{a_i} q'$.

The correctness of Sink Combine transformation is easy to see: the states $q_1, ..., q_k$ can be replaced by a single state because they all are non-initial and have one incoming transition originating from the same state bearing the same label.

Source Combine. Let $q_1, ..., q_k$ be some non-final states of A, all having exactly one outgoing transition with the same label a_i to a state q of A where q is different from all q_l, $l \in \{1, ..., k\}$. Then $q_1, ..., q_k$ can be combined into one state q', meaning that $q_1, ..., q_k$ and their incoming and outgoing transitions are removed and replaced by a new non-final state q' which is an initial state if and only if any of $q_1, ..., q_k$ is an initial state, with all incoming transitions of $q_1, ..., q_k$ now entering q', and the transition $q' \xrightarrow{a_i} q$.

The correctness of Source Combine transformation is obvious by the equivalence of the states $q_1, ..., q_k$.

These four transformations are schematically presented in Figure 5.1.

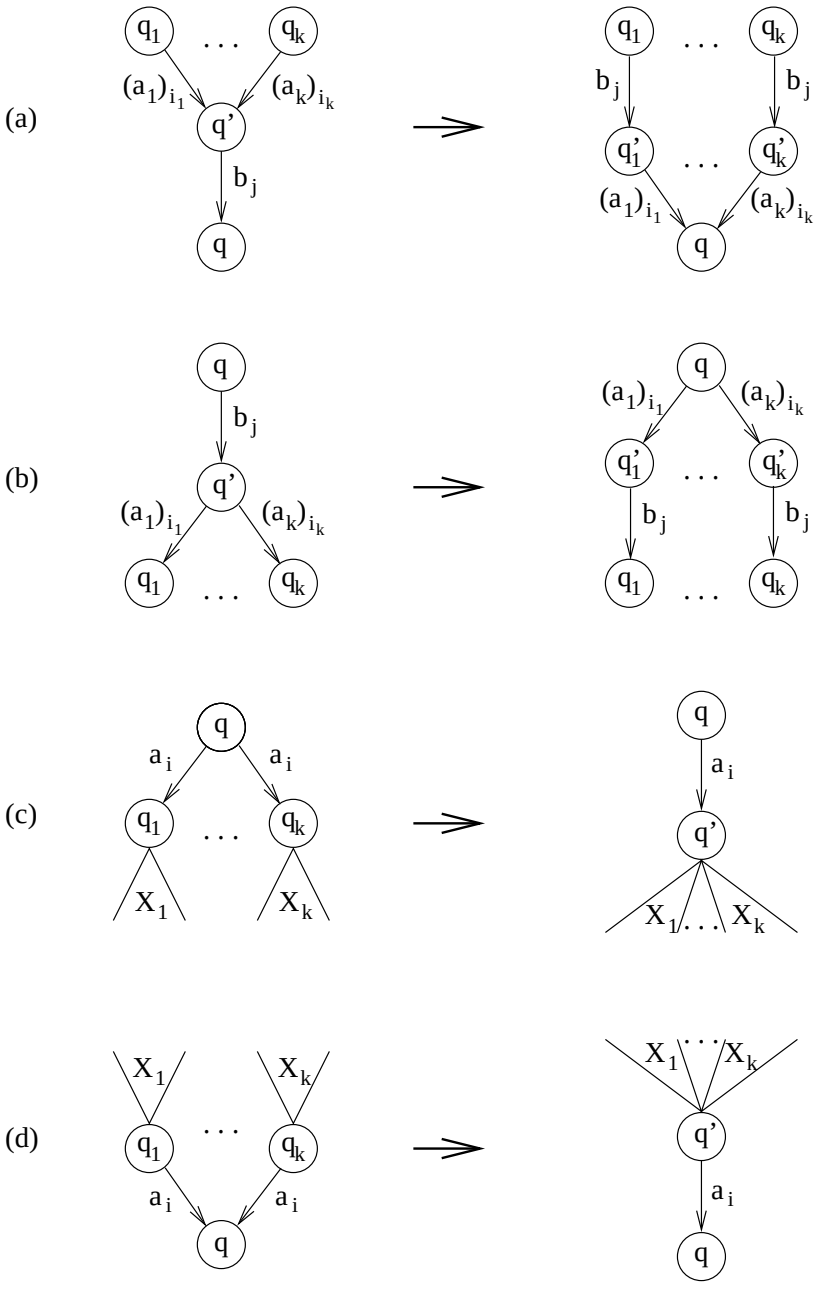

Figure 5.1: Automaton transformations: (a) Swap Upwards; (b) Swap Downwards; (c) Sink Combine; (d) Source Combine.

procedure MoveTransitionUp($A, (q_1, a_i, q_2), q$**)**
1. **if** transition (q_1, a_i, q_2) exists in A **then**
2. use the Sink Combine transformation to merge all such states that are reachable from q_1 by a transition labelled by a_i and suitable for this transformation;
3. **if** $q \neq q_1$ and $outdegree(q_1) = 1$ **then**
4. use the Swap Upwards transformation on the outgoing transition of q_1 and let T be the set of transitions with the label a_i created by this transformation;
5. **for all** $(q'_1, a_i, q'_2) \in T$ where $q'_1, q'_2 \in Q$ **do**
6. `MoveTransitionUp`$((q'_1, a_i, q'_2), q)$;

Figure 5.2: Procedure `MoveTransitionUp()`

5.2 Reduction algorithm

Based on the four automaton transformations presented in Section 5.1, we have designed an algorithm to reduce the size of an n-tape automaton $A = (Q, \Sigma, \delta, I, F)$.

Let us first present a central part of the algorithm which is the procedure `MoveTransitionUp()` shown in Figure 5.2, and the conditions to apply this procedure.

The procedure `MoveTransitionUp()` gets the automaton A, a transition (q_1, a_i, q_2) and some state q of A as its input parameters. The goal of this procedure is to decrease the number of states and transitions of A by "moving" the transition (q_1, a_i, q_2) in the automaton graph "up", applying Swap Upwards and Sink Combine transformations on the way, until this transition, along with one or more other transitions which have the same label, will be replaced by a transition out of q (instead of q_1).

Below, we will specify a set of conditions which guarantee that applying the procedure `MoveTransitionUp()` is possible and leads to the reduction of the number of states and transitions of the automaton.

Definition 5.1 *Let $q \in Q$ and $i \in \{1, ..., n\}$. A transition is called a* future transition *of a state q concerning tape i if it is the first transition involving this tape on some path in A that starts from q.*

Let us fix some $q \in Q$, $a \in \Sigma$ and $i \in \{1, ..., n\}$. We want to find a set of future transitions of q concerning tape i, with the label a_i, such that by calling the procedure `MoveTransitionUp()` for each of these transitions and the state q, we can reduce the number of states of A by a certain

amount.

If we denote a set of future transitions of q concerning tape i bearing the label a_i, by $ft_{q,i,a}$, then let us denote the set of all paths in A, which start from q and end by any transition in $ft_{q,i,a}$, by $P_{ft_{q,i,a}}$. Consider the following conditions imposed on the path set $P_{ft_{q,i,a}}$. Let p be a path in the set $P_{ft_{q,i,a}}$ and let the two last states on p be denoted by q' and q''. Then the conditions are as follows:

(i) there are no loops in p, except that q'' may be equal to q;

(ii) every state on p that appears after q and before q'' is non-initial and non-final, all of its incoming and outgoing transitions are traversed by some path in $P_{ft_{q,i,a}}$, and all of its incoming transitions involve a tape that is different from i;

(iii) if q' has more than one outgoing transition then q'' is non-initial and has only one incoming transition.

Proposition 5.1 *Let $q \in Q$, $a \in \Sigma$ and $i \in \{1, ..., n\}$. Then there is a unique maximal set $FT_{q,i,a}$ of future transitions of q concerning tape i, with the label a_i, such that the conditions (i) – (iii) hold for the set $P_{FT_{q,i,a}}$.*

Proof. Consider the set $ft^{all}_{q,i,a}$ of all future transitions of q concerning tape i, with the label a_i. If $ft^{all}_{q,i,a}$ is an empty set then the proposition is trivially true, with $FT_{q,i,a}$ being empty as well. Now, let us assume that the set $ft^{all}_{q,i,a}$ is not empty. Then we partition $ft^{all}_{q,i,a}$ into non-intersecting non-empty subsets $ft_1, ..., ft_k$ in the following way. Consider any two transitions t and u in $ft^{all}_{q,i,a}$ with the corresponding path sets $P_{\{t\}}$ and $P_{\{u\}}$ consisting of paths starting from q and ending by t and u, respectively. Let t and u belong to the same subset $ft_j, j \in \{1, ..., k\}$, if and only if there exists a pair of paths $p_t \in P_{\{t\}}$ and $p_u \in P_{\{u\}}$ such that p_t and p_u have a common state that is different from the starting and the ending states of both p_t and p_u.

For every $j \in \{1, ..., k\}$, let us consider the set of paths P_{ft_j} that start from q and end by any transition belonging to ft_j. It is easy to see by the definition of the sets $ft_j, j \in \{1, ..., k\}$, that if the conditions (i) – (iii) hold for some path sets $P_{ft_{j_1}}, ..., P_{ft_{j_l}}$ where $j_m \in \{1, ..., k\}, m \in \{1, ..., l\}$, then these conditions are also true for the union of these path sets.

But for a set of future transitions that, for some $j \in \{1, ..., k\}$, contains as a subset a non-empty proper subset ft'_j of ft_j but not the whole ft_j, the corresponding set of paths, beginning from q and ending by such a transition, does not satisfy the condition (ii). This is because then there must be a state r on a path starting from q and ending by some transition

in ft'_j such that some outgoing transition of r does not belong to this path whereas it belongs to some path from q ending by some transition in ft_j.

To summarize, a maximal subset of $ft^{all}_{q,i,a}$ such that the conditions (i)–(iii) hold for all paths starting from q and ending by any transition in this subset, is a union of all those $ft_j, j \in \{1, ..., k\}$, such that the conditions (i)–(iii) hold for P_{ft_j}. This set is uniquely defined. □

Let $FT_{q,i,a}$ be the maximal set of future transitions of q concerning tape i, with the label a_i, such that the conditions (i) – (iii) hold for the set of all paths in A which start from q and end by any transition belonging to $FT_{q,i,a}$. Then the following proposition holds.

Proposition 5.2 *The series of calls to the procedure* `MoveTransitionUp()` *shown in Figure 5.2 where it is called with every transition in* $FT_{q,i,a}$ *and* q*, results in size reduction of* A *by* $|FT_{q,i,a}| - 1$ *states. Also, at least the same number of transitions are eliminated from* A *by this process.*

Proof. Consider the series of calls to the procedure `MoveTransitionUp()` as specified in the proposition. Let us denote by $FT'_{q,i,a}$ the set of transitions, which initially consists of all transitions in $FT_{q,i,a}$, and which is modified according to the changes that the above-mentioned calls to the procedure `MoveTransitionUp()` produce in the automaton. That is, those transitions in $FT'_{q,i,a}$ that are removed from the automaton by Sink Combine and Swap Upwards transformations are also removed from $FT'_{q,i,a}$, and all new transitions bearing the label a_i that are created by the same transformations to replace the removed transitions, are added to the set $FT'_{q,i,a}$. Using the same notation as above, let $P_{FT'_{q,i,a}}$ be the set of all paths in A which start from q and end by any transition belonging to $FT'_{q,i,a}$. Based on changes that the Sink Combine and Swap Upwards transformations can make in the automaton, it is not difficult to see that the conditions (i)–(iii) hold for all paths in $P_{FT'_{q,i,a}}$ after any number of Sink Combine and/or Swap Upwards transformations have been performed during the `MoveTransitionUp()` calls under consideration.

When the procedure `MoveTransitionUp()` is called for a given transition $(q_1, a_i, q_2) \in FT'_{q,i,a}$ and the state q, it first checks if the transition still exists, and if yes, then it uses the Sink Combine transformation to merge such states reachable from q_1 by a transition labelled by a_i that satisfy the conditions for this operation. If there are at least two such states to merge then, by (iii), this merging concerns every state that is reachable from q_1 by any transition in $FT'_{q,i,a}$.

If q_1 is different from q and if there is only one transition leaving q_1 then

by (ii), the Swap Upwards transformation can be applied to this transition, followed by a set of recursive calls to `MoveTransitionUp()` for transitions labelled by a_i that were created by the Swap Upwards transformation. In case $q_1 \neq q$ and q_1 has more than one outgoing transition, all of these transitions belong to (one or more) paths in $P_{FT'_{q,i,a}}$ each of which end by a transition belonging to $FT'_{q,i,a}$. When considering these transitions forming a subset of $FT'_{q,i,a}$, we claim that when `MoveTransitionUp()` is called for each of the transitions in this subset, then these calls to `MoveTransitionUp()` finally eliminate all outgoing transitions of q_1 and replace them by a single transition for which the Swap Upwards transformation can be applied. This is because, during this process, there is always some transition in the above-mentioned subset of $FT'_{q,i,a}$, for which either Sink Combine or Swap Upwards transformation can be applied, or otherwise some of the conditions (i)–(iii) cannot hold.

The conditions (i)-(iii) guarantee that the process involving the series of calls to `MoveTransitionUp()` as stated in the proposition concerns only the states that are on some path of $P_{FT'_{q,i,a}}$, terminates and replaces the transitions in $FT_{q,i,a}$ by a single transition originating from q with the label a_i (although q can still have other outgoing transitions with this label that do not belong to $FT_{q,i,a}$).

During this process, $|FT_{q,i,a}| - 1$ states and at least as many transitions are eliminated, as shown next. When the Sink Combine transformation is applied to merge some states reachable from q_1 by a transition labelled by a_i then along with the mergeable states it eliminates the same number of transitions originating from q_1. Also, as the merged state (as well as any other state) may have at most one transition with any label to any state, those outgoing transitions of the merged state that would otherwise become duplicates are eliminated as well. The Swap Upwards transformation creates the equal number of new states and transitions with the label a_i going to these states, thus increasing the number of states and transitions by the same amount. Therefore, as totally $|FT_{q,i,a}|$ transitions are replaced by a single transition, $|FT_{q,i,a}| - 1$ states and at least as many transitions are eliminated by the process. □

Let us suppose that we find the maximal sets of future transitions for a state q and tape i, for all possible labels, such that the corresponding path set of each of these transition sets satisfies the conditions (i)–(iii). The next proposition ensures that the number of states eliminated from the automaton by applying the series of `MoveTransitionUp()` calls as in Proposition 5.2, for each of these sets, is independent of the order in which

the sets are handled.

Proposition 5.3 *Let $q \in Q$, $a, b \in \Sigma$ and $i \in \{1, ..., n\}$. Let $FT_{q,i,a}$ and $FT_{q,i,b}$ be the maximal sets of future transitions of q concerning tape i, labelled a_i and b_i, respectively, with their corresponding path sets $P_{FT_{q,i,a}}$ and $P_{FT_{q,i,b}}$ which satisfy the conditions (i)–(iii) as in Proposition 5.1. Let us first apply the transformations described in Proposition 5.2 for the set $FT_{q,i,a}$. After that, Proposition 5.2 still holds for the set $FT_{q,i,b}$.*

Proof. First, we claim that for any path pair $p_t \in P_{FT_{q,i,a}}$ and $p_u \in P_{FT_{q,i,b}}$, if p_t and p_u have a common state other than q then it is the ending state of both p_t and p_u. Indeed, if we suppose that there is a common state r on paths p_t and p_u such that $r \neq q$ and r is not the ending state of p_t or p_u, then the condition (ii) must be violated. Therefore, such state cannot exist.

Based on this observation, the transformations performed for the set $FT_{q,i,a}$ as described in Proposition 5.2 do not interfere with the transformations for the set $FT_{q,i,b}$. Therefore, the proposition holds. □

Similarly to the conditions (i)–(iii), symmetric conditions can be specified that allow to eliminate states from the automaton by a procedure that uses the Source Combine and Swap Downwards transformations.

The algorithm for reducing the size of A is presented in Figure 5.3. The algorithm uses a variable m to indicate the number of states eliminated from A. First, the algorithm calls a procedure `CombineInitialStates()` which merges those initial states of A that do not have any incoming transitions, into one single state, analogously to the Sink Combine transformation, and returns the number of states eliminated this way. Then, a similar procedure named `CombineFinalStates()`, combining into a single state those accepting states of A that have no outgoing transitions, and analogous to the Source Combine transformation, is called. The value of m is updated on return of both of these procedures. Then, a copy of A is made, denoted by A_1.

Next, the idea is that for each tape of A, as many states as possible are eliminated from A using a procedure `Upwards()` (presented in Figure 5.4), and from A_1 using a similar procedure `Downwards()`. Given the automaton tape *tape*, the procedure `Upwards()` finds for each state q a set $FT_{q,tape}$ that is the union of all maximal sets $FT_{q,tape,a}$ of future transitions of the state q concerning the tape *tape* and some symbol a, such that the conditions (i)–(iii) are satisfied for the path set $P_{FT_{q,tape,a}}$. For all $FT_{q,tape,a}$ that consist of at least two transitions, a state q' is found which has the same

```
Reduce A
1.   m := 0;
2.   m := m+ CombineInitialStates(A);
3.   m := m+ CombineFinalStates(A);
4.   A_1 := CopyOf(A);
5.   reduced := true;
6.   while reduced = true do
7.        reduced := false;
8.        for tape := 1 to n do
9.             m_up := Upwards(A, tape);
10.            m_down := Downwards(A_1, tape);
11.            if m_up > 0 or m_down > 0 then
12.                 if m_up ≥ m_down then
13.                      A_1 := CopyOf(A);
14.                      m := m + m_up;
15.                 else
16.                      A := CopyOf(A_1);
17.                      m := m + m_down;
18.                 reduced := true;
19.  return A, m;
```

Figure 5.3: Reduction algorithm

set of future transitions for this tape and symbol, $FT_{q',tape,a} = FT_{q,tape,a}$, such that the conditions (i)–(iii) are satisfied for $P_{FT_{q',tape,a}}$, and which is as close to the transitions in $FT_{q,tape,a}$ as possible. Then the procedure `MoveTransitionUp()` is called for all of the transitions in $FT_{q',tape,a}$ and q', and by Proposition 5.2 the value of m is decreased by $|FT_{q',tape,a}| - 1$. After considering every such set $FT_{q,tape,a}$, the loop over all states is started again. This process continues until no further reductions of A can be achieved using this approach for any state of A. The return value of `Upwards()` indicates the number of states eliminated by it. The procedure `Downwards()` acts similarly in a symmetric fashion.

In case any states were eliminated from either A or A_1, a smaller one of these automata is retained and the next round with a next tape is performed using two copies of that automaton. Also, the value of m is updated accordingly. This process is continued until no more states are eliminated for any tape. Finally, the algorithm outputs the (possibly) reduced automaton A, and the total number of eliminated states m.

procedure Upwards$(A, tape)$
1. $m := 0$;
2. $reduced := true$;
3. **while** $reduced = true$ **do**
4. $reduced := false$;
5. **for all** $q \in Q$ as long as $reduced = false$ **do**
6. find a set $FT_{q,tape} = \bigcup_{a \in \Sigma' \subseteq \Sigma} FT_{q,tape,a}$ such that for each $a \in \Sigma'$, $FT_{q,tape,a}$ is as in Proposition 5.1;
7. **for all** $a \in \Sigma'$ where $|FT_{q,tape,a}| > 1$ **do**
8. find a state q' such that $FT_{q',tape,a} = FT_{q,tape,a}$, $FT_{q',tape,a}$ is as in Proposition 5.1, and the longest path from q' to the originating state of any transition in $FT_{q,tape,a}$ is of minimal length;
9. **for all** $t \in FT_{q',tape,a}$ **do**
10. `MoveTransitionUp`(A, t, q');
11. $m := m + |FT_{q',tape,a}| - 1$;
12. $reduced := true$;
13. return m;

Figure 5.4: Procedure `Upwards()`

5.3 Analysis of the reduction algorithm

Let A be an n-tape automaton with N states over an alphabet Σ. In the following sections we show the correctness of the reduction algorithm and analyze its time and space complexity. We also show that the algorithm does not increase the number of transitions in the automaton.

5.3.1 Correctness of the algorithm

Let us denote the automaton produced by the reduction algorithm applied to A by A_{red}.

We consider the reduction algorithm to be *correct* if, when applied to an automaton A, it produces A_{red} such that $|A_{red}| < |A|$ and $L(A_{red}) = L(A)$ or $A_{red} = A$. That is, the correct reduction algorithm either modifies the automaton so that the number of states of the automaton decreases without changing the language accepted by the automaton, or alternatively, does not change the automaton at all.

The procedures `CombineInitialStates()` and `CombineFinalStates()` obviously either reduce the size of the automaton by combining some of

its states together, or do not produce any changes to the automaton. The transformations performed by these procedures do not change the language accepted by the automaton. These facts along with Propositions 5.2 and 5.3 guarantee that the reduction algorithm is correct.

5.3.2 Time complexity

In this section we show that the reduction algorithm when applied to A with n tapes, N states and alphabet Σ takes time $O(n^3|\Sigma|^3N^4)$.

Let us denote $S = |\Sigma|$. First, `CombineInitialStates()` takes time $O(n^2S^2N^3)$ as there can be $O(N)$ initial states to eliminate, each of which having $O(nSN)$ outgoing transitions and each of these transitions will be checked against the $O(nSN)$ outgoing transitions of the "combined" initial state before adding into its transition list, or deleting from A in additional $O(nSN)$ time. Similarly, `CombineFinalStates()` takes time $O(n^2S^2N^3)$, too.

Copying A into another automaton can be done in $O(nSN^2)$ time as there are at most N states and nSN^2 transitions in A.

In the procedure `Upwards()`, finding a set $FT_{q,tape}$ (line 6) involves traversing each transition of A at most a constant number of times, thus takes totally $O(nSN^2)$ time. Finding a state q' (line 8) can be achieved in $O(N)O(nSN^2) = O(nSN^3)$ time. A series of `MoveTransitionUp()` calls with all transitions in $FT_{q',tape,a}$ takes $O(n^2S^2N^3)$ time, as at most $O(nSN)$ work is done with each of the $O(nSN^2)$ transitions of A by the Sink Combine and Swap Upwards transformations. Thus, the `for` loop of `Upwards()` of lines 5–13 takes time $O(N)O(nSN^2) + O(S)O(n^2S^2N^3) = O(n^2S^3N^3)$. Totally, this loop is run for $O(nN)$ times because there can be $O(N)$ such loop runs that reduce the size of the automaton and for each such run there can be $O(n)$ runs that do not result in reduction.

As a similar reasoning can be applied to the procedure `Downwards()`, and as the total time to run the `if` command on lines 11–18 of the reduction algorithm is $O(N)O(n)O(nSN^2) = O(n^2SN^3)$, the total time taken by the reduction algorithm is $O(n^3S^3N^4)$.

5.3.3 Space complexity

In the following we show that a space requirement of the reduction algorithm when applied to A is $O(n|\Sigma|N^2)$.

As in the previous section, let us denote $S = |\Sigma|$. First, as there are N states and $O(nSN^2)$ transitions in A then a space needed to store A is $O(nSN^2)$. In addition to A, the algorithm maintains a copy A_1 of A

which takes $O(nSN^2)$ space, too. No additional space is needed to run the procedures `CombineInitialStates()` and `CombineFinalStates()`.

Next, let us discuss a space requirement of the procedure `Upwards()`. Computing the set $FT_{q,tape}$ and the state q' may involve $O(N)$ space for active recursive procedure calls. To store the set $FT_{q,tape}$, $O(SN^2)$ space is needed. During a series of `MoveTransitionUp()` calls initiated with the transitions of $FT_{q',tape,a}$, not more than $O(nSN)$ new states and transitions per each state of A may be created and thus a space requirement of A remains $O(nSN^2)$ at any time during the series. There are $O(nSN^2)$ `MoveTransitionUp()` calls in this series, but only $O(N)$ calls are active at the same time. As each active call needs $O(nSN)$ space to store a set of transitions T created by `SwapUpwards()` then the amount of space this series of `MoveTransitionUp()` calls requires is $O(nSN^2)$.

Therefore, the space requirement for `Upwards()` is $O(nSN^2)$. The same space limit holds for the procedure `Downwards()`. Thus the total space requirement of the reduction algorithm is $O(nSN^2)$.

Corollary 5.1 *For a fixed number of tapes and fixed alphabet, the time and space complexities of the reduction algorithm are $O(N^4)$ and $O(N^2)$, respectively.*

5.3.4 The effect of the algorithm on the number of transitions

Proposition 5.4 *As a result of applying the reduction algorithm on an automaton, the number of transitions in the automaton either decreases or remains the same.*

Proof. Neither the procedure `CombineInitialStates()` nor the procedure `CombineFinalStates()` increases the number of transitions in the automaton. By Proposition 5.2, the procedure `Upwards()` does not do this either, and the same applies for `Downwards()`. Therefore, the above statement holds. □

5.4 Example

In this section, we present an example of automata on which the reduction algorithm works nicely.

Let $L_k = \{ww^R \mid w \in \{0,1\}^k\}$ where $k \geq 1$ be a set of strings consisting of concatenations of any binary string of length k and its reversal string. Let L_k^* be the set that consists of strings obtained by concatenating zero

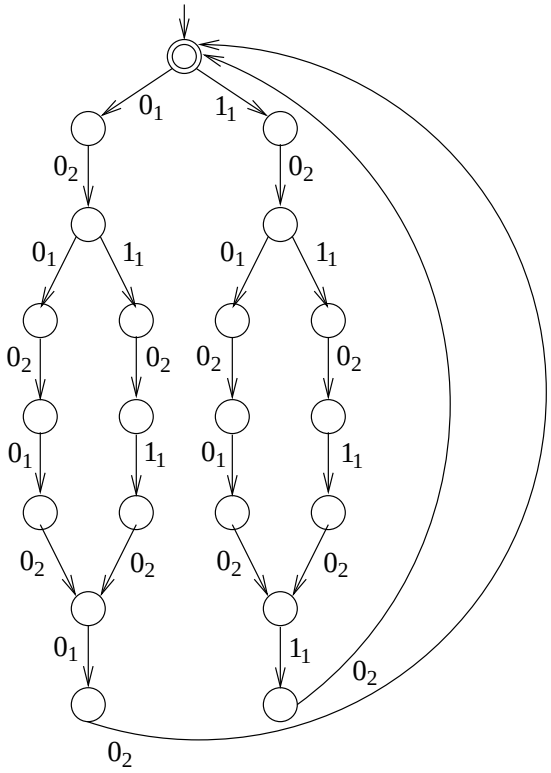

Figure 5.5: 2-tape automaton A_2 constructed from the leftmost automaton of Figure 3.4

or more times the elements of L_k. For any $k \geq 1$, let A_k be the 2-tape automaton constructed in the following way. We start with a minimal 1-tape automaton A_k^1 accepting the language L_k^* (as shown in Figure 3.4 (at left) for $k = 2$), and modify it to a 2-tape automaton A_k by creating for every transition $q_1 \xrightarrow{a} q_2$ of A_k^1, where q_1 and q_2 are some states of A_k^1 and $a \in \{0, 1\}$, a new state q_1' and a new transition $q_1' \xrightarrow{0_2} q_2$, and replacing the transition $q_1 \xrightarrow{a} q_2$ with the transition $q_1 \xrightarrow{a_1} q_1'$. The resulting automaton for $k = 2$ is shown in Figure 5.5. The automaton A_k accepts all such tuples (w_1, w_2) where w_1 is a string belonging to the set L_k^* and w_2 is a string of the same length as w_1, consisting of only 0s.

The automaton A_k constructed in this way is not a minimal automaton accepting its language. The size of A_k can be reduced by applying the reduction algorithm to this automaton. In the following we discuss how the reduction algorithm works on the automaton A_2.

First, we notice that neither the procedure `CombineInitialStates()` nor `CombineFinalStates()` changes the given automaton which has only one initial and one final state. Then consider the procedure `Upwards()` called for the first tape. By this procedure, the automaton states are considered one after another, and future transition sets of each state concerning the first tape as specified in Proposition 5.1 are computed. As there is no state q and symbol $a \in \{0, 1\}$ such that the maximal future transition set for q, concerning the first tape and symbol a, would consist of more than one transition, no calls to the procedure `MoveTransitionUp()` are issued. So, the return value of `Upwards()` is 0, and similarly for `Downwards()`. Now, let us consider the call to `Upwards()` for the second tape. As every transition in A_2 concerning the second tape is labelled 0_2, we only find the future transitions bearing this label for the second tape. The algorithm does not specify in which order the automaton states are considered, but let us start with the initial state which we denote by q_0. Then the set of future transitions $FT_{q_0,2,0}$ consists of two outgoing transitions of these two states into which q_0 has transitions. The corresponding path set $P_{FT_{q_0,2,0}}$ consists of two paths, both of which satisfy the conditions (i)–(iii) as can be easily verified. The state q' as specified on line 8 of `Upwards()` is the same as q_0. By calling the procedure `MoveTransitionUp()` for one of the transitions in $FT_{q_0,2,0}$, this transition will be "moved up" by Swap Upwards transformation, but no more transformations are induced by the recursive call to the procedure. Then, `MoveTransitionUp()` is called for the other transition in $FT_{q_0,2,0}$, which means that the Swap Upwards transformation is applied to this transition followed by the Sink Combine transformation applied to the resulting transition by the recursive call to `MoveTransitionUp()`. The automaton after these transformations is depicted in Figure 5.6.

Next in `Upwards()`, the loop over the automaton states, where their future transitions involving the second tape are computed, starts all over again. The automaton after the procedure `Upwards()` stops is also the resulting automaton of the whole reduction algorithm because by applying `Downwards()` instead of `Upwards()`, the same number of states are eliminated.

When we apply the reduction algorithm on A_k, the result is an automaton A_{kred} of a smaller size as shown in Figure 5.7 for the case $k = 2$. Obviously, A_{kred} is a minimal automaton. However, it is not the only minimal automaton accepting its language. Actually, if we prefer the "downward" swaps to the "upward" swaps in a situation where the number of eliminated states both ways is the same, that is, if we modify the reduction algorithm so that line 12 would be "**if** $m_{up} > m_{down}$ **then**" instead of "**if**

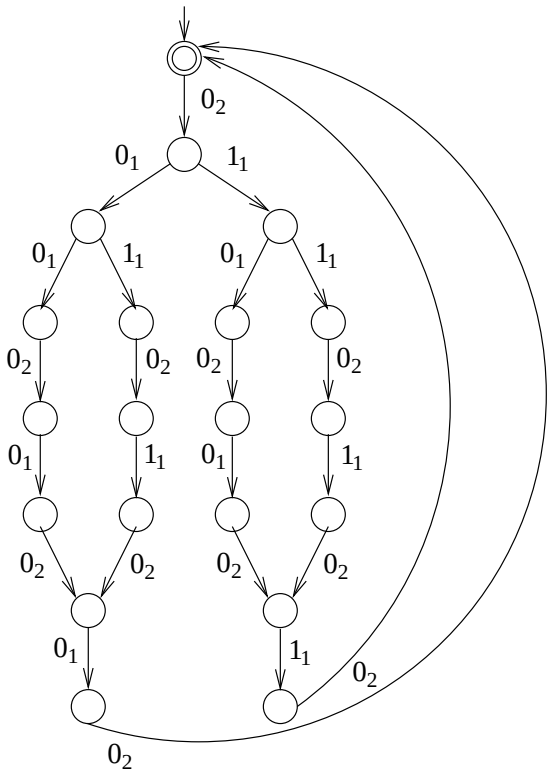

Figure 5.6: Intermediate step in the application of the reduction algorithm to A_2

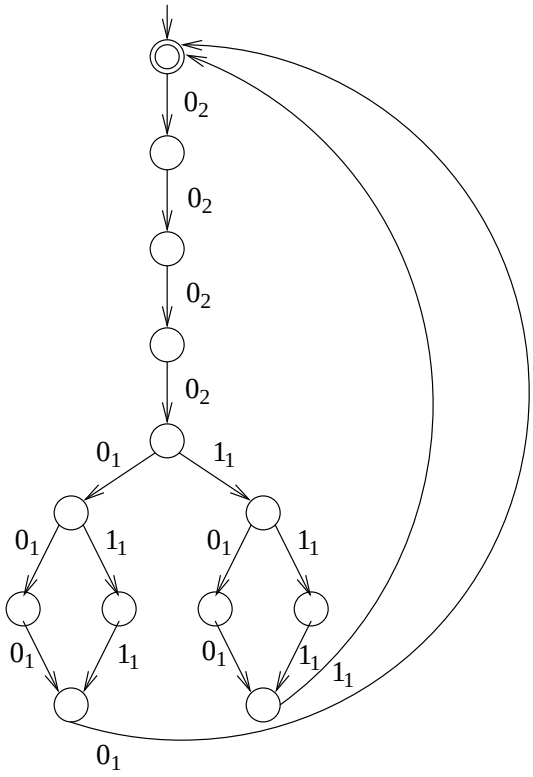

Figure 5.7: The automaton A_{2red} obtained by applying the reduction algorithm on A_2

$m_{up} \geq m_{down}$ **then**", then the reduced automaton would be different as shown in Figure 5.8 for the case $k = 2$, but (in this example) still have the same number of states.

In the general case, though, the reduction algorithm does not necessarily produce a minimal automaton. An example of this kind of automaton is presented in Section 6.3.

For any $k \geq 1$, the automaton A_k has $7 \times 2^k - 7$ states and $8 \times 2^k - 8$ transitions. In the reduced automaton A_{kred} there are $3 \times 2^k + 2k - 3$ states and $4 \times 2^k + 2k - 4$ transitions. Thus, $4 \times 2^k - 2k - 4$ states and the same number of transitions are eliminated from A_k by the reduction algorithm.

We have performed experiments with the reduction algorithm on the automata A_k where $k = 1, ..., 7$. The experiments have been carried out on an Intel Pentium 4 1.8GHz server computer. The results of these experiments are presented as a table in Figure 5.9. The table shows the size of A_k,

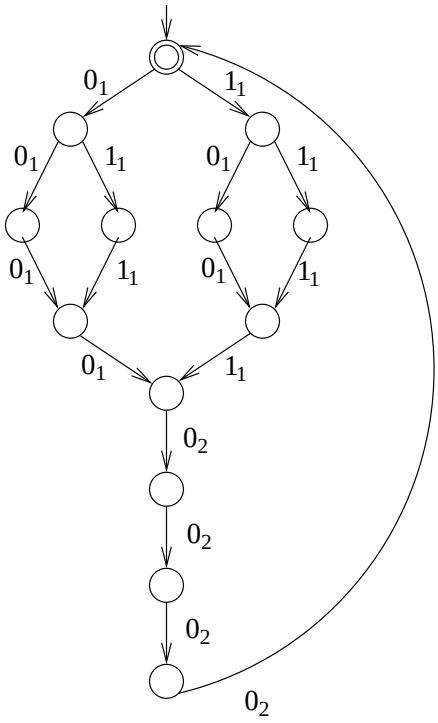

Figure 5.8: The automaton obtained by applying the modified reduction algorithm on A_2

k	$\lvert A_k \rvert$	$\lvert A_k \rvert - \lvert A_{kred} \rvert$	time
1	7	2	0.000
2	21	8	0.001
3	49	22	0.003
4	105	52	0.021
5	217	114	0.147
6	441	240	1.121
7	889	494	8.542

Figure 5.9: A table presenting the sizes of A_k, the numbers of eliminated states and the average times taken by the reduction algorithm

the number of eliminated states from A_k, and the user CPU time in seconds averaged over 100 calls to the procedure implementing the reduction algorithm, for $k = 1, ..., 7$.

Chapter 6

Application of the reduction algorithm

In this chapter we apply the multitape-automata size-reduction algorithm of Chapter 5 in the string database system described in [9]. In fact, the first motivation to develop the reduction algorithm was given by the development of this system. This database system has been discussed in several earlier papers as well, such as [13]–[15]. However, the most recent and most complete overview of the system is given in [9]. In Section 6.1 we present the basis of the system called Alignment Declaration language and in Section 6.2 we describe how expressions in this language can be compiled to and viewed as multitape automata. An example of a string operation expressed in the Alignment Declaration language with the corresponding automata is presented in Section 6.3. In Section 6.4 we discuss the application of the reduction algorithm combined with the one-tape DFA minimization procedure on this kind of automata. Finally, in Section 6.5 we present experimental results of applying this approach on a set of string operations.

6.1 Alignment Declaration language

In this section we describe the Alignment Declaration language that is the basis of our string database system. This language is based on Alignment Calculus [10]. The Alignment Declaration language is designed to describe string comparison and manipulation operations. A string operation expressed in this language is called an *alignment declaration*. We give a definition of an expression in the Alignment Declaration language recursively as follows.

First, a basic statement of this language is an *on*-statement. It consists of the word `on`, which is possibly preceded by a *scan part* and possibly followed by a *condition part*. A scan part starts with the word `scan` or `rightscan` followed by a list of string variables corresponding to strings this scan part has an effect on. Depending on whether the scan part starts with a `scan` or a `rightscan`, the effect of the scan part is to move the position of a currently considered character of the corresponding string to the next or the previous position. A condition part is a Boolean combination of character comparisons written as `x='a'`, `x=y`, `x=[` or `x=]`. These comparisons evaluate true if, respectively, the currently considered character of a string denoted by variable `x` is `'a'`, the same as the currently considered character of a string denoted by variable `y`, the left endmarker, or the right endmarker. The left and right endmarkers, respectively, denote the positions before the first and after the last character of a string. Initially, the current position for any string considered is the left endmarker. An `on`-statement holds if and only if, after taking into account possible changes of currently considered characters of the strings pointed out by the scan part, the condition part evaluates true.

An `on`-statement is an expression in the Alignment Declaration language. Other expressions are defined recursively as follows. If Φ_1 and Φ_2 are expressions then their concatenation $\Phi_1\Phi_2$ is an expression, `repeat * times` Φ_1 `end` is an expression, and `choose` $\Phi_1|\Phi_2$ `end` is an expression. The expression $\Phi_1\Phi_2$ holds if and only if Φ_1 holds and Φ_2 holds when evaluated starting from the same currently considered character positions where the evaluation of Φ_1 ends. The expression `repeat * times` Φ_1 `end` holds if and only if a k-fold concatenation of Φ_1 with itself holds for some $k \geq 0$. The expression `choose` $\Phi_1|\Phi_2$ `end` holds if and only if Φ_1 holds or Φ_2 holds.

Some additional constructs are defined in the Alignment Declaration language to make the expressions shorter. For example, `repeat * times scan x on x='a' end` can be written as `scan* x on x='a'`, and similarly for `rightscan`. Also, successive `on`-statements with their scan part starting with `scan` and involving a single variable like `scan x on x='a' scan x on x='c' scan x on x='a'` can be replaced by a statement `read x on 'aca'`. In a similar way, `on`-statements with their scan part starting with `rightscan` can be replaced by a `rightread` statement.

By default, it is assumed that all characters in all strings belong to the 8-bit ASCII character set. However, subsets of the ASCII alphabet can be defined using an `alphabet` declaration, for example, like `alphabet DNA : 'A', 'C', 'G', 'T'`, which defines an alphabet named DNA with the four specified symbols. This kind of named alphabets can be used, for

example, in `keep`-statements like `keep x in DNA ... end` where in place of the dots there is a list of statements that form a scope in which one wants to limit the content of the string corresponding to `x` to the DNA characters. This is equivalent to adding to the condition part of every `on`-statement in that scope involving `x` an extra condition of the form `(x='A' or x='C' or x='G' or x='T' or x=[ or x=])`. It is assumed that each alignment declaration is surrounded by implicit `keep`-statements for every variable used in the declaration, which limit the content of all strings to the ASCII characters. The condition part of each `on`-statement that involves some variable is affected by both implicit and explicit `keep`-statements for that variable in whose scope the `on`-statement is.

All alignment declarations start with a name of the string operation, followed by a parenthesized list of the string variables that this string operation uses, separated by commas.

There is also a macro call facility in the language and a possibility of adding extra annotations to the query result which we will not discuss here. The complete syntax of the Alignment Declaration language can be found in [9].

6.2 Alignment declarations as multitape automata

Let Φ be an alignment declaration with string variables $x_1, ..., x_n$. In the following we describe how Φ can be translated into an n-tape automaton A with ϵ-transitions in the two-way mixed-state model.

First, every Boolean formula in all `on`-statements of Φ is transformed so that it consists of only `and` and `or` operations combining character comparisons in the forms `x='a'`, `x=[` or `x=]`. For this reason, first rules such as `not (`Φ_1 `and` Φ_2`)` $\equiv$ `not` Φ_1 `or not` Φ_2 and `not (`Φ_1 `or` Φ_2`)` $\equiv$ `not` Φ_1 `and not` Φ_2 as well as `not not` $\Phi_1 \equiv \Phi_1$ are applied when possible. After this, `not` operations and character comparisons in the form `x=y` are eliminated, using string alphabets that are valid for the `on`-statements under consideration.

To create A, we use a function `Compile()` described below which takes either an alignment declaration or a part of it as its first input argument and the automaton state as its second input argument, possibly creates new states and transitions into the automaton and calls itself recursively, and finally outputs an automaton state.

At the beginning, let A consist of a single final state q_F. Then, a call to the function `Compile(`Φ, q_F`)` builds up A and yields the initial state q_I of A. Let Φ_1 and Φ_2 denote either expressions in the Alignment Declaration

language or parts of such Boolean formulas described above. Let q be an automaton state. In addition to the special symbols `[` and `]` denoting the left and right endmarkers, the symbol `@` is used to denote any ASCII character or the right endmarker, `L` is used to denote a tape movement to the left by one position, and `R` is used to denote a tape movement to the right by one position. Then we define the function `Compile()` by induction over the structure of the alignment declaration as follows:

1) `Compile(`$\Phi_1\Phi_2, q$`)` = `Compile(`Φ_1 `and` Φ_2, q`)` = `Compile(`Φ_1, `Compile(`Φ_2, q`))`;
2) `Compile(choose` $\Phi_1|\Phi_2$ `end,` q`)` = `Compile(`Φ_1 `or` Φ_2, q`)` $= q_1$ where q_1 is a new state that has ϵ-transitions to `Compile(`Φ_1, q`)` and `Compile(`Φ_2, q`)`;
3) `Compile(repeat * times` Φ_1 `end,` q`)` $= q_1$ where q_1 is a new state that has ϵ-transitions to q and `Compile(`Φ_1, q_1`)`;
4) `Compile(on` Φ_1, q`)` = `Compile(`Φ_1, q`)`;
5) `Compile(scan` $x_{i_1}, ..., x_{i_k}$ `on` Φ_1, q`)`$= q_1$ where $i_j \in \{1, ..., n\}$ and $q_1, ..., q_k$ are new states with transitions $q_j \xrightarrow{\texttt{L}_{i_j}} q_{j+1}$ for $j = 1, ..., k$, with q_{k+1} = `Compile(`Φ_1, q`)`;
6) `Compile(rightscan` $x_{i_1}, ..., x_{i_k}$ `on` Φ_1, q`)` $= q_1$ where $i_j \in \{1, ..., n\}$ and $q_1, ..., q_{2k}$ are new states with transitions $q_{2j-1} \xrightarrow{\texttt{@}_{i_j}} q_{2j}$, $q_{2j} \xrightarrow{\texttt{R}_{i_j}} q_{2j+1}$, and $q_{2j-1} \xrightarrow{\texttt{[}_{i_j}} q_{2j+1}$ for $j = 1, ..., k$, where q_{2k+1} = `Compile(`Φ_1, q`)`;
7) `Compile(`$x_i = \sigma, q$`)` $= q_1$ where $i \in \{1, ..., n\}$, σ is either an ASCII character or the left or right endmarker, and q_1 is a new state with a transition $q_1 \xrightarrow{\sigma_i} q$;
8) `Compile(`$true, q$`)` $= q_1$ where q_1 is a new state with an ϵ-transition to q;
9) `Compile(`$false, q$`)` $= q_1$ where q_1 is a new state with no transitions.

Note that the compilation of an `on`-statement depends on whether its scan part starts with the word `scan` or `rightscan`. If it starts with `rightscan`, the tape movement to the right is preceded by a check whether the current character on the given tape is any ASCII character or the right endmarker, in which case the tape move will take place; in case the current character is the left endmarker, the tape move will not occur. These checks are explicitly put into the automaton by means of corresponding transitions. If an `on`-statement starts with the word `scan`, then, in principle, we could use similar reasoning and put similar extra transitions into the automaton. However, this is not necessary because, initially, the current

character of each tape is the left endmarker in which case the tape movement to the left is possible, and such transitions in the automaton implying moving the tape to the left in the situation where the current tape character is the right endmarker, are replaced by ϵ-transitions by a later analysis of the automaton as discussed below.

As a result of the function call `Compile(`Φ, q_F`)` a two-way multitape automaton A with one initial and one final state is created. This automaton can be considered as another representation of the alignment declaration Φ. Besides the ϵ-transitions, there are two kinds of transitions in A: those that represent the character checks of the input strings, and those that represent the tape movements to the left and right.

The ϵ-transitions can be eliminated from the automaton, using Propositions 2.1 and 2.2 in Section 2.3 and known methods from the one-tape automata theory.

Next, the automaton A can be modified to eliminate redundant checks and tape movements from it. For this reason, the automaton is expanded so that in each state it remembers the last transition labels for all tapes which appeared on any path from the initial state to the given state. Based on this information, the redundant transitions are replaced with ϵ-transitions, and such transitions that obviously cannot be applied are eliminated from the automaton. For example, if the last transition concerning tape i was labelled by $\mathtt{a}_i$ where `a` is some ASCII character, and the current transition is labelled with $\mathtt{@}_i$ then the current transition can be considered redundant and is therefore replaced by an ϵ-transition. Also, a transition labelled $\mathtt{L}_i$ after a transition with the label $\mathtt{]}_i$ is replaced by an ϵ-transition, and similarly for a transition labelled $\mathtt{R}_i$ after a transition with the label $\mathtt{[}_i$. Or, if the last transition was labelled $\mathtt{L}_i$ and the current transition is labelled $\mathtt{[}_i$ then it is obvious that the current transition is not possible to take and it can be eliminated.

After the expansion, the ϵ-transitions are eliminated from the automaton. Also, the states that are not on any path from an initial state to a final state are eliminated from the automaton.

6.3 An example

This section presents an example of an alignment declaration describing a property involving two strings x and y from the alphabet $\{a, b\}$ where y is the reversal of x.

The alignment declaration is as follows:

```
reversal(x, y)
   keep x in 'a', 'b'
   keep y in 'a', 'b'
      scan* x on
      scan x on x=]
      repeat * times
         rightscan x on
         scan y on x=y
      end
      rightscan x on x=[
      scan y on y=]
   end
   end
```

The corresponding 2-tape automaton, obtained as the result of applying the function `Compile()` on this alignment declaration where the ϵ-transitions are eliminated, is shown in Figure 6.1. Here, the first tape corresponds to variable x and the second one to y.

The expanded version of the automaton where ϵ-transitions and non-useful states are eliminated, is shown in Figure 6.2. Let us denote this automaton $A_{reversal}$.

6.4 Reducing the size of a multitape automaton

Our first motivation to develop the multitape automata size reduction algorithm of Section 5.2 was to apply it to the multitape automata corresponding to the alignment declarations as described above. Applying this algorithm to two-way multitape automata is possible, based on the discussion in Section 2.3. To continue with the example of the previous section, if we apply the reduction algorithm to the automaton $A_{reversal}$ in Figure 6.2, the size of the automaton is reduced from 23 states to 16 states. The resulting automaton denoted by $RED(A_{reversal})$ is shown in Figure 6.3.

Interestingly, another method to try to reduce the size of a multitape automaton is to interpret it as a one-tape automaton as discussed in Section 2.3, find a minimal DFA equivalent to this one-tape automaton, and then interprete the resulting automaton again as a multitape automaton. Although, generally, finding an equivalent minimal DFA for a given one-tape nondeterministic automaton can result in a larger automaton instead of a smaller one, this approach is worth trying on the automata produced

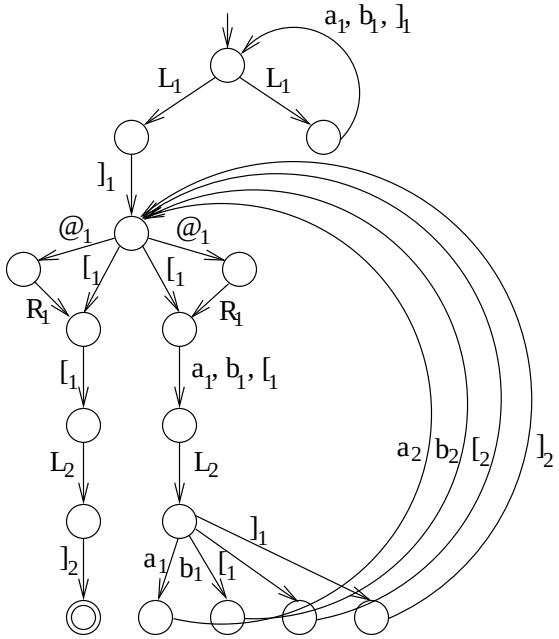

Figure 6.1: The automaton corresponding to the alignment declaration `reversal(x, y)` after eliminating ϵ-transitions

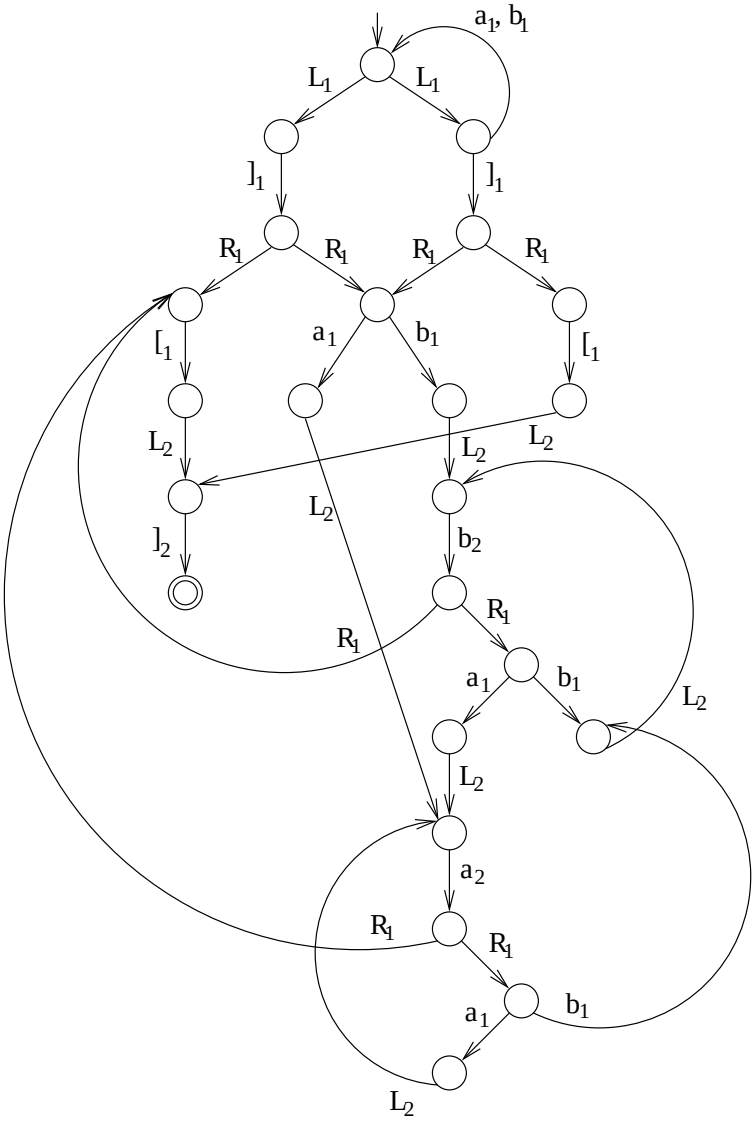

Figure 6.2: The automaton $A_{reversal}$

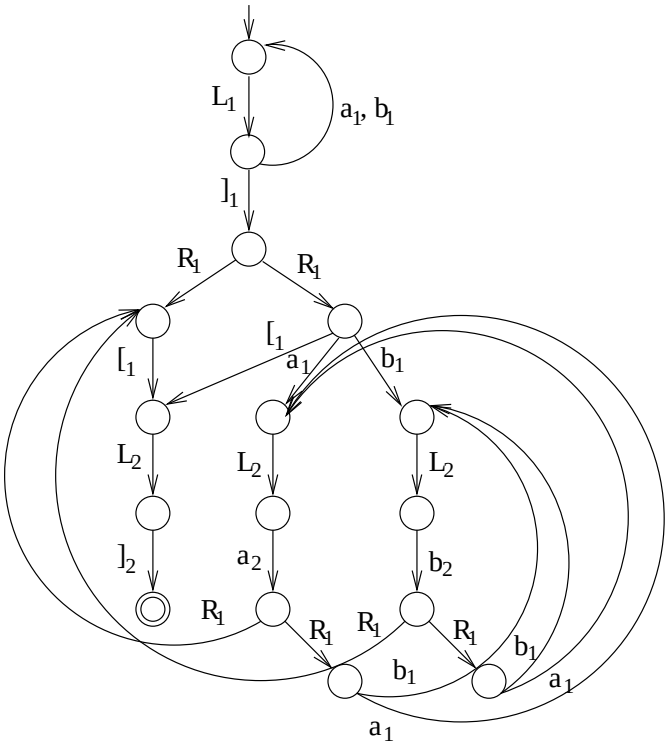

Figure 6.3: The automaton $RED(A_{reversal})$

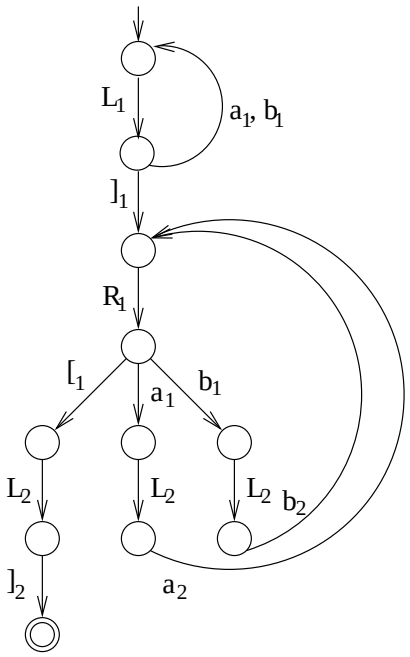

Figure 6.4: The automaton $MIN(A_{reversal})$

from the alignment declarations as the test results presented in the next section indicate.

If we apply this method to the automaton $A_{reversal}$ of our example, the result is the automaton $MIN(A_{reversal})$ with only 11 states as shown in Figure 6.4.

Now, applying the reduction algorithm after the one-tape DFA minimization can lead to a further size reduction of the multitape automaton. In the current example, if we apply the reduction algorithm to $MIN(A_{reversal})$ then the result is even smaller automaton $RED(MIN(A_{reversal}))$ with 9 states as shown in Figure 6.5. Further application of the one-tape DFA minimization on this automaton does not change the automaton.

Similarly, we can apply the one-tape DFA minimization procedure on the automaton $RED(A_{reversal})$. The resulting automaton is the same as the automaton $MIN(A_{reversal})$. Applying the reduction algorithm again on this automaton, we obtain the same automaton as $RED(MIN(A_{reversal}))$.

Concerning this example, we can notice two things. First, the one-tape DFA minimization reduces the size of the automaton of this example. Second, the end result of applying the reduction algorithm and the one-tape

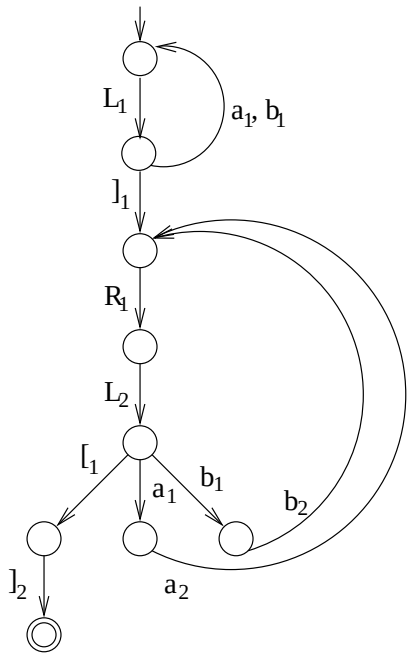

Figure 6.5: The automaton $RED(MIN(A_{reversal}))$

DFA minimization procedure one after another until no more size reduction of the automaton can be achieved, does not depend on which of these two procedures was applied first.

However, in general, the DFA minimization does not necessarily reduce the size of the automaton. Also, generally, the end result of applying the aforementioned two algorithms one after another does depend on which one is applied first, as the results of the experiments show in the next section.

Finally, we propose the following algorithm to reduce the size of a multitape automaton A that alternatingly applies two size-reducing algorithms. Apply two sequences of algorithms consisting of the one-tape DFA minimization procedure and our reduction algorithm of Chapter 5 by turn on A, at one time starting with the DFA minimization algorithm and the other time starting with our reduction algorithm, and stopping whenever no more changes happen in A, or alternatively, the size of A is increased by the DFA minimization procedure. Output the smaller of the resulting two automata.

6.5 Experimental results

To test our reduction algorithm as presented at the end of the previous section, we have considered a set of alignment declarations expressing different string properties, and made experiments with the corresponding multitape automata. Besides the *reversal* operation discussed in Section 6.3, we have considered the following string predicates.

The *substring*, *subsequence*, *prefix* and *suffix* operations applied on some strings x and y as written below respectively express the property that x is a substring, a subsequence, a prefix or a suffix of y. As the declarations indicate, both strings belong to the alphabet $\{a, b\}$.

```
substring(x, y)
   keep x in 'a', 'b'
   keep y in 'a', 'b'
      scan* y on not y=]
      scan* x,y on x=y
      scan x on x=]
   end
   end
```

```
subsequence(x, y)
   keep x in 'a', 'b'
   keep y in 'a', 'b'
      repeat* times
         choose
            scan x,y on x=y and not y=] |
            scan y on not y=]
         end
      end
      scan x on x=]
   end
   end


prefix(x, y)
   keep x in 'a', 'b'
   keep y in 'a', 'b'
      scan* x, y on x=y
      scan x on x=]
   end
   end


suffix(x, y)
   keep x in 'a', 'b'
   keep y in 'a', 'b'
      scan* x on
      scan x on x=]
      scan* y on
      scan y on y=]
      rightscan* x, y on x=y
      rightscan x on x=[
   end
   end
```

The *concatenation* and *shuffle* operations applied on strings x, y and z as written below correspondingly express the property that x is a concatenation or a shuffle of y and z. As above, the strings belong to the alphabet $\{a, b\}$.

```
concatenation(x, y, z)
   keep x in 'a', 'b'
   keep y in 'a', 'b'
   keep z in 'a', 'b'
      scan* x,y on x=y and not y=]
      scan y on y=]
      scan* x,z on x=z and not x=]
      scan x,z on x=] and z=]
   end
   end
   end


shuffle(x, y, z)
   keep x in 'a','b'
   keep y in 'a','b'
   keep z in 'a','b'
      repeat* times
         choose
            scan x, y on x=y and not x=] |
            scan x, z on x=z and not x=]
         end
      end
      scan x, y, z on x=] and y=] and z=]
   end
   end
   end
```

The *overlap* operation applied on strings x, y and z expresses the property that the suffix of x is the same that the prefix of y for $|z|$ or more characters. Again, the alphabet of the strings is $\{a, b\}$.

```
overlap(x, y, z)
   keep x in 'a', 'b'
   keep y in 'a', 'b'
   keep z in 'a', 'b'
      scan* x on not x=]
      scan* x,y,z on x=y
      scan x,z on x=] and z=]
   end
   end
   end
```

The operation *edit_distance* applied on strings x, y and z expresses the property that the edit distance of x and y is not greater than $|z|$. Here the strings belong to the DNA alphabet $\{a, c, g, t\}$.

```
edit_distance(x, y, z)
   keep x in 'a','c','g','t'
   keep y in 'a','c','g','t'
   keep z in 'a','c','g','t'
      repeat* times
         choose
            scan x, y on x=y |
            scan x, y, z on not z=] |
            scan x, z on not z=] |
            scan y, z on not z=]
         end
      end
      scan x, y on x=] and y=]
   end
   end
   end
```

We have made experiments by applying the reduction algorithm described at the end of Section 6.4 on the multitape automata generated from the alignment declarations described above. The algorithm is applied to the expanded automata. The results of the experiments are presented

in the form of a table in Figure 6.6. For each string predicate considered, the table shows the number of tapes n and the alphabet size $|\Sigma|$ of the corresponding automaton, the size of the original automaton $|A_{orig}|$ (the result of applying the function `Compile()` on the corresponding alignment declaration) after eliminating ϵ-transitions from it, the size of the expanded automaton $|A_{exp}|$ after ϵ-transition elimination. The reduction algorithm is applied on the ϵ-transition-free expanded automaton A_{exp} of each string operation. The table shows the size of the automaton during the reduction process, given in two rows: the upper row shows the automaton size in the reduction sequence where the one-tape DFA minimization procedure is applied first, and the lower row shows the automaton size in the sequence where the reduction algorithm of Chapter 5 is applied first. The numbers in the columns with the word MIN and RED indicate the size of the automaton in the reduction process, after applying the DFA minimization procedure or the reduction algorithm of Chapter 5, respectively.

In many cases, both of these reduction sequences end up with the automata of the same size, although the resulting automata are not necessarily identical. However, sometimes one or the other of these approaches produces a smaller automaton, as is the case with the *suffix* and *overlap* operations.

Interestingly, in all of these examples, applying the DFA minimization algorithm never increases the size of the automaton. On the contrary, applying this algorithm seems to achieve most of the reduction of the automaton size.

For most of the string predicates considered in our experiments, the size of the reduced automaton is smaller than the size of the corresponding original automaton. However, this is not always the case, as indicated by the automata of the *overlap* and *edit_distance* predicates. Based on this, one may question the usefulness of the whole expansion and reduction process. However, if one has in mind the efficiency of simulating the computations of automata, then avoiding redundant checks of tape symbols and those paths that are not possible to follow, seem to be important. Fortunately, most of the size growth in the expanded automata seems to disappear as a result of the reduction process.

String predicate	n	$\|\Sigma\|$	$\|A_{orig}\|$	$\|A_{exp}\|$	Automaton size during the reduction process				
reversal	2	2	17	23	MIN 11	RED 9	MIN 9		
					RED 16	MIN 11	RED 9	MIN 9	
substring	2	2	11	18	MIN 9	RED 9			
					RED 11	MIN 9	RED 9		
subsequence	2	2	11	17	MIN 7	RED 7			
					RED 12	MIN 7	RED 7		
prefix	2	2	9	16	MIN 7	RED 7			
					RED 9	MIN 7	RED 7		
suffix	2	2	18	25	MIN 11	RED 11			
					RED 17	MIN 13	RED 13		
concatenation	3	2	21	20	MIN 13	RED 12	MIN 12		
					RED 19	MIN 13	RED 12	MIN 12	
shuffle	3	2	21	51	MIN 12	RED 10	MIN 10		
					RED 45	MIN 12	RED 10	MIN 10	
overlap	3	2	15	48	MIN 21	RED 20	MIN 20		
					RED 29	MIN 19	RED 19		
edit distance	3	4	24	168	MIN 27	RED 27			
					RED 134	MIN 30	RED 28	MIN 27	RED 27

Figure 6.6: Automaton sizes before and during the reduction process

Chapter 7

Conclusions

In this thesis we have considered some issues related to minimization of one-tape automata and size reduction of multitape automata.

Concerning the much-researched topic of minimization of one-tape automata, it is a well-known fact that a minimal NFA can be exponentially smaller than the equivalent minimal DFA. However, quite often, the sizes of a minimal NFA and DFA of a given language do not differ that much or are even the same. Here, our interest has been to find conditions under which a minimal DFA is also a minimal NFA of its language. We have shown that in addition to the earlier known fact of bideterministic automata being minimal among DFAs, these automata are uniquely minimal among NFAs as well. We have also shown that bideterministic automata have a minimal number of transitions. Also, we have specified a set of sufficient conditions guaranteeing that a minimal DFA of a given language or the reversal of a minimal DFA of the reversal language is a minimal NFA accepting that language. The latter result is more general and, in fact, the minimality of bideterministic automata can be obtained from that result as a special case.

Interesting future research topics in this area could be to find other automata classes besides bideterministic automata with unique minimal NFAs (if they exist), and to specify other classes of automata with a minimal number of transitions. More generally, minimality issues in respect to the number of transitions seem to be not so well studied so far.

Concerning multitape automata, we have shown that bideterminism in multitape automata does not guarantee minimality of these automata. Still, a bideterministic multitape automaton is a unique minimal automaton among all automata with the same set of accepting computations.

We have developed a polynomial-time size reduction algorithm for one-way multitape automata and shown its good size-reduction effect on an

example automata family. We have applied this algorithm along with the one-tape DFA minimization procedure to two-way multitape automata corresponding to example string predicates defined in the string-manipulating database system of [9], with good results.

However, there may be ways to improve the reduction algorithm. For example, applying an approach similar to that described in [20] and [21] where the states of an NFA are merged according to their largest left and right invariant equivalences, could be useful for the size reduction of multitape automata as well.

References

[1] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley 1986.

[2] Angluin, D. Inference of reversible languages. *Journal of the ACM* **29**, 3 (1982), 741–765.

[3] Arnold, A., Dicky, A., and Nivat, M. A note about minimal nondeterministic automata. *Bull. European Assoc. Theoret. Comput. Sci.* **47** (1992), 166–169.

[4] Brzozowski, J.A. Canonical regular expressions and minimal state graphs for definite events. In *Proceedings of the Symposium on Mathematical Theory of Automata*, MRI Symposia Series, vol. 12, Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1963, 529–561.

[5] Champarnaud, J.-M., and Coulon, F. NFA reduction algorithms by means of regular inequalities. In *Proceedings of DLT 2003*, Lecture Notes in Computer Science 2710, Springer, 2003, 194–205.

[6] Elgot, C.C., and Mezei, J.E. On relations defined by generalized finite automata. *IBM J. Res. Develop.* **9**, (1965), 47–68.

[7] Fischer, P.C., and Rosenberg, A.L. Multitape one-way nonwriting automata. *J. Comput. System Sci.* **2**, (1968), 88–101.

[8] Glaister, I., and Shallit, J. A lower bound technique for the size of nondeterministic finite automata. *Inform. Proc. Letters* **59**, (1996), 75–77.

[9] Grahne, G., Hakli, R., Nykänen, M., Tamm, H., and Ukkonen, E. Design and implementation of a string database query language. *Information Systems* **28**, (2003), 311–337.

[10] Grahne, G., Nykänen, M., and Ukkonen, E. Reasoning about strings in databases. *J. Comput. System Sci.* **59**, (1999), 116–162.

[11] Griffiths, T.V. The unsolvability of the equivalence problem for λ-free nondeterministic generalized machines. *J. ACM*, **15**, 3 (1968), 409–413.

[12] Grigorieff, S. Modelization of deterministic rational relations. *Theoretical Computer Science*, **281**, (2002), 423–453.

[13] Hakli, R., Nykänen, M., and Tamm, H. A declarative programming system for manipulating strings. In *Sixth Fenno-Ugric Symposium on Software Technology*, Sagadi, Estonia, 1999, 29–40.

[14] Hakli, R., Nykänen, M., and Tamm, H. Adding string processing capabilities to data management systems. In *Proceedings of the Seventh International Symposium on String Processing and Information Retrieval (SPIRE 2000)*, IEEE Computer Society Press, Silver Spring, MD, 2000, 122–131.

[15] Hakli, R., Nykänen, M., Tamm, H., and Ukkonen, E. Implementing a declarative string query language with string restructuring. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, Lecture Notes in Computer Science 1551, Springer, 1999, 179–195.

[16] Harju, T., and Karhumäki, J. The equivalence problem of multitape finite automata. *Theoretical Computer Science*, **78**, (1991), 347–355.

[17] Hopcroft, J. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical Report CS-190, Stanford University, 1971.

[18] Hopcroft, J.E., Motwani, R. and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley 2001.

[19] Hopcroft, J.E., and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley 1979.

[20] Ilie, L., and Yu, S. Algorithms for computing small NFAs. In *Proceedings of MFCS 2002*, Lecture Notes in Computer Science 2420, Springer, 2002, 328–340.

[21] Ilie, L., and Yu, S. Reducing NFAs by invariant equivalences. *Theoretical Computer Science*, **306**, 2003, 373–390.

[22] Ilie, L., Navarro, G., and Yu, S. On NFA reductions. In *Theory is forever*, Lecture Notes in Computer Science 3113, Springer, 2004, 112–124.

[23] Jiang, T., and Ravikumar, B. Minimal NFA problems are hard. *SIAM J. Comput.* **22**, 6 (1993), 1117–1141.

[24] Kameda, T., and Weiner, P. On the state minimization of nondeterministic automata. *IEEE Trans. Comput.* **C-19**, 7 (1970), 617–627.

[25] Karhumäki, J. Applications of finite automata. In *Proceedings of MFCS 2002*, Lecture Notes in Computer Science 2420, Springer, 2002, 40–58.

[26] Khachatryan, V.E. Complete system of equivalent transformations for multitape automata. *Programming and Computer Software* **29**, 1, (2003), 43–54.

[27] Matz, O., and Potthoff, A. Computing small nondeterministic finite automata. In *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, BRICS Notes Series, May 1995, 74–88.

[28] Muder, D.J. Minimal trellises for block codes. *IEEE Trans. Inform. Theory* **34**, 5 (1988), 1049–1053.

[29] Pin, J.-E. On reversible automata. In *Proceedings of the first LATIN conference*, Lecture Notes in Computer Science 583, Springer, 1992, 401–416.

[30] Rabin, M.O., and Scott, D. Finite automata and their decision problems. *IBM J. Res. Develop.* **3**, (1959), 114–125.

[31] Sengoku, H. Minimization of nondeterministic finite automata. Master's thesis, Kyoto University, 1992.

[32] Shankar, P., Dasgupta, A., Deshmukh K., and Rajan B.S. On viewing block codes as finite automata. *Theoretical Computer Science*, **290**, 3 (2003), 1775–1797.

[33] Tamm, H., and Ukkonen, E. Bideterministic automata and minimal representations of regular languages. In *Proceedings of the CIAA 2003*, Lecture Notes in Computer Science 2759, Springer, 2003, 61–71.

[34] Tamm, H., and Ukkonen, E. Bideterministic automata and minimal representations of regular languages. *Theoretical Computer Science*, **328**, 1–2 (2004), 135–149.

[35] Watson, B.W. Taxonomies and toolkits of regular language algorithms. PhD dissertation, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.

[36] Yamasaki, H. On multitape automata. In *Proceedings of MFCS'79*, Lecture Notes in Computer Science 74, Springer, 1979, 533–541.

TIETOJENKÄSITTELYTIETEEN LAITOS
PL 68 (Gustaf Hällströmin katu 2 b)
00014 Helsingin yliopisto

DEPARTMENT OF COMPUTER SCIENCE
P.O. Box 68 (Gustaf Hällströmin katu 2 b)
FIN-00014 University of Helsinki, FINLAND

JULKAISUSARJA **A**

SERIES OF PUBLICATIONS **A**

Reports may be ordered from: Kumpula Science Library, P.O. Box 64, FIN-00014 University of Helsinki, FINLAND.

A-2001-1 J. Rousu: Efficient range partitioning in classification learning. 68+74 pp. (Ph.D. thesis)

A-2001-2 M. Salmenkivi: Computational methods for intensity models. 145 pp. (Ph.D. thesis)

A-2001-3 K. Fredriksson: Rotation invariant template matching. 138 pp. (Ph.D. thesis)

A-2002-1 A.-P. Tuovinen: Object-oriented engineering of visual languages. 185 pp. (Ph.D. thesis)

A-2002-2 V. Ollikainen: Simulation techniques for disease gene localization in isolated populations. 149+5 pp. (Ph.D. thesis)

A-2002-3 J. Vilo: Discovery from biosequences. 149 pp. (Ph.D. thesis)

A-2003-1 J. Lindström: Optimistic concurrency control methods for real-time database systems. 111 pp. (Ph.D. thesis)

A-2003-2 H. Helin: Supporting nomadic agent-based applications in the FIPA agent architecture. 200+17 pp. (Ph.D. thesis)

A-2003-3 S. Campadello: Middleware infrastructure for distributed mobile applications. 164 pp. (Ph.D. thesis)

A-2003-4 J. Taina: Design and analysis of a distributed database architecture for IN/GSM data. 130 pp. (Ph.D. thesis)

A-2003-5 J. Kurhila: Considering individual differences in computer-supported special and elementary education. 135 pp. (Ph.D. thesis)

A-2003-6 V. Mäkinen: Parameterized approximate string matching and local-similarity-based point-pattern matching. 144 pp. (Ph.D. thesis)

A-2003-7 M. Luukkainen: A process algebraic reduction strategy for automata theoretic verification of untimed and timed concurrent systems. 141 pp. (Ph.D. thesis)

A-2003-8 J. Manner: Provision of quality of service in IP-based mobile access networks. 191 pp. (Ph.D. thesis)

A-2004-1 M. Koivisto: Sum-product algorithms for the analysis of genetic risks. 155 pp. (Ph.D. thesis)

A-2004-2 A. Gurtov: Efficient data transport in wireless overlay networks. 141 pp. (Ph.D. thesis)

A-2004-3 K. Vasko: Computational methods and models for paleoecology. 176 pp. (Ph.D. thesis)

A-2004-4 P. Sevon: Algorithms for Association-Based Gene Mapping. 101 pp. (Ph.D. thesis)

A-2004-5 J. Viljamaa: Applying Formal Concept Analysis to Extract Framework Reuse Interface Specifications from Source Code. 206 pp. (Ph.D. thesis)

A-2004-6 J. Ravantti: Computational Methods for Reconstructing Macromolecular Complexes from Cryo-Electron Microscopy Images. 100 pp. (Ph.D. thesis)

A-2004-7 M. Kääriäinen: Learning Small Trees and Graphs that Generalize. 45+49 pp. (Ph.D. thesis)

A-2004-8 T. Kivioja: Computational Tools for a Novel Transcriptional Profiling Method. 98 pp. (Ph.D. thesis)