Czech Technical University in Prague

Faculty of Electrical Engeneering

Congruences on Finite Automata

Bachelor Thesis

Study programme: Open Informatics
Branch of study: Computer and Information Science

Thesis advisor: prof. RNDr. Marie Demlová, CSc.

Nela Grimová

Prague, 2015

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Nela G r i m o v á

Study programme: Open Informatics

Specịalisation: Computer and Information Science

Title of Bachelor Project: Congruences on Finite Automata

Guidelines:

1. Study congruences on finite automata with output and properties of Myhill-Nerode's Theorem for finite automata.
2. Choose an appropriate algorithm for constructing congruences and implement it.
3. Based on experiments evaluate the implemented algorithm.

Bibliography/Sources:

[1] M. Demlová, V. Koubek: Algebraická teorie automatů, SNTL, Praha, 1990
[2] M. Mohri: Minimization algorithms for sequential transducers, Theoretical Computer Science, 2000
[3] A. Maletti: Myhill-Nerode theorem for sequentioal tranducers over unique GCD-monoids, 2005

Bachelor Project Supervisor: prof. RNDr. Marie Demlová, CSc.

Valid until: the end of the summer semester of academic year 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 14, 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Nela G r i m o v á

Studijní program: Otevřená informatika (bakalářský)

Obor: Informatika a počítačové vědy

Název tématu: Kongruence na konečných automatech

Pokyny pro vypracování:

1. Seznamte se s kongruencemi konečných automatů s výstupem a s vlastnostmi Myhill-Nerodovy věty pro tyto automaty.
2. Vyberte vhodný algoritmus pro výpočet kongruencí a implementujte ho.
3. Na experimentech proveďte hodnocení implementovaného algoritmu.

Seznam odborné literatury:

[1] M. Demlová, V. Koubek: Algebraická teorie automatů, SNTL, Praha, 1990
[2] M. Mohri: Minimization algorithms for sequential transducers, Theoretical Computer Science, 2000
[3] A. Maletti: Myhill-Nerode theorem for sequentioal tranducers over unique GCD-monoids, 2005

Vedoucí bakalářské práce: prof. RNDr. Marie Demlová, CSc.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 14. 1. 2015

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Declaration

I declare that I have written my bachelor thesis myself and used only the sources listed in **References** in accordance with the methodical observance of ethical principles in the preparation of university thesis.

Prague, 22th May 2015 Nela Grimová

Acknowledgement

I wish to express my sincere thanks to my thesis advisor prof. RNDr. Marie Demlová, CSc. for all her help, advice, time and comments.

Abstrakt

Cílem té to práce je seznámení s kongruencemi na konečných automatech, a to zejména s elementární stavovou kongruencí. Jsou zde popsány algoritmy na konstrukci této kongruence a následně jsou diskutovány jejich výsledky na příkladech. Dále je popsán zobecněný případ elementární stavové kongruence – nejmenší kongruence obsahující relaci.

Klíčová slova: konečné automaty; stavová kongruence; elementární stavová kongruence; faktorový automat, nejmenší kongruence obsahující relaci.

Abstract

The aim of this thesis is an introduction to congruences on finite automata, especially to elementary automaton state congruences. Two algorithms for finding elementary state automaton congruences are described and their result on various examples are discussed. Further, a generalization of elementary state automaton congruences is described – the smallest state automaton congruences containing a relation.

Keywords: finite automata; automaton state congruence; elementary automaton state congruence; factor automaton; the smallest automaton congruences contating a relation.

Contents

List of Figures

List of Tables

1 Preface

The theory of finite-state machines is a widely studied theme, which has had a big influence to the development of computers and the study of computability and the processing of formal languages. Although, a large area of knowledge about finite-state machines is already covered, there exist various topics that need a closer look. A study of automaton state congruences is one of them.

In the second chapter, we will introduce basic definitions and facts covering words, equivalence relations and automata. Automaton state congruences are the theme of the third chapter. We will focus especially on elementary automaton state congruences, these are the smallest automatata congruences merging two distinct states, and their construction.

In the fourth chapter, we will introduce the algorithm for finding elementary automaton state congruences and its accelerated version and then we will prove correctness and the worst processing time of the algorithm.

In the next chapter, we will describe implementation of both versions of this algorithm, which is used in the following chapter for testing the algorithm on various automata. The reader can find measured values and graphs in **Appendix**.

In the last chapter, we will generalize our approach to the following problem: Given a relation R on the set of states Q of an automaton, we would like to construct the smallest automaton state congruence containg R.

2 Preliminaries

2.1 Alphabet, Words over an Alphabet, Language

An **alphabet** X is a finite, nonempty set of symbols.

A **word over an alphabet** X is a finite sequence of symbols of X. If a word does not contain a single element, we will call it the **empty word** and donote it by ϵ.

A **length of a word** w is the number of its symbols; we denote it by $|w|$. The empty word has length zero.

Further, X^* indicates the set of all words over the alphabet X, and X^+ is the set of all nonempty words over the alphabet X.

A **concatenation of words** is the operation on X^* denoted as $\cdot$ and defined by: if $w \in X^*$, $v \in X^*$, $w = x_1x_2 \dots x_n$, $v = y_1y_2 \dots y_m$ (n is length of w and m is length of v), then $w \cdot v = x_1x_2 \dots x_ny_1y_2 \dots y_m$.

Concatenation is associative ($(u \cdot v) \cdot w = u \cdot (v \cdot w)$) for all $u, v, w \in X^*$ and ϵ is its neutral element.

In this text, an alphabet will be the set of input symbols X, and the set of output symbols Y.

All these definitions are paraphrases from (see [1], p. 28-29).

A **language** L is a set of words over an alphabet X, $L \subseteq X^*$ (see [1], p. 30).

2.2 Binary Relation

A binary relation on a set A is a subset of the carthesian product $A \times A$. A binary relation R on the set A is called:

- **reflexive**: if $\forall a \in A$, it holds that $a\,R\,a$ (every element from A is in relation with itself)
- **symmetric**: when $\forall a, b \in A$, it holds that if $a\,R\,b$, then $b\,R\,a$
- **antisymmetric**: when $\forall a, b \in A$, it holds that if $a\,R\,b$ and $b\,R\,a$, then $a = b$
- **transitive**: when $\forall a, b, c \in A$, it holds that if $a\,R\,b$ and $b\,R\,c$, then $a\,R\,c$

2.3 Equivalence Relation

A relation R on a set A is called an equivalence relation if it is reflexive, symmetric and transitive.

2.3.1 Equivalence Classes

The equivalence class of R containing an element $a \in A$ is defined as:

$$[a]_R = \{b \in X; a\, R\, b\}$$

A quotient set A/R of a set A, determined by the equivalence relation R, is defined as the set of all equivalence classes of R, i.e.:

$$A/R = \{[a]_R; a \in A\}$$

The quotient set A/R forms a decomposition of A. Indeed, each $a \in A$ belongs to $[a]_R$, and the classes are pairwise disjoint.

As long as it is not misleading, we will write $[a]$ instead of $[a]_R$.

Note that two special kinds of an equivalence relation always exist:

- Δ is the identical equivalence in which no distinct elements are merged, so every element corresponds to one equivalence class.
- ∇ is the universal equivalence relation which merges all elements of a set A, so it has only one equivalence class.

If an equivalence R on A is distinct from Δ and ∇, we call it a nontrivial congruence on A.

2.4 Finite-state Machine

A finite-state machine (or a finite-state automaton) is a theoretical model of computation consisting of a finite set of states, a finite set of input symbols, a finite set of output symbols, a transition function and an output function. According to the form of the output function, finite-state machines are divided into:

2.4.1 Mealy Automaton

A Mealy automaton is a quintuple (Q, X, Y, δ, α)[see 2], where

- Q is a finite nonempty set of states
- X is a finite nonempty set of input symbols
- Y is a finite nonempty set of output symbols
- δ (transition function) is a mapping δ: $Q \times X \to Q$
- α (output function) is a mapping α: $Q \times X \to Y$

2.4.2 Moore Automaton

A Moore automaton is a quintuple (Q, X, Y, δ, β), where Q, X, Y and δ have the same meaning as above, and the output function β is a mapping β: $Q \to Y$.

2.4.3 State Automaton

Let M be either a Mealy or a Moore automaton. Then a corresponding state automaton is the triple (Q, X, δ).

2.5 Notation for Automata

There are two commonly used and preferred notations for describing automata ([1], p. 48-49):

2.5.1 Transition Table

A transition table is a tabular represention of the transition function δ and the output function α or β. The rows of the table correspond to the states of the automaton and the columns correspond to the inputs. In the case of a Mealy automaton, the entry for the state q and the input x is a pair $\delta(q, x)/\alpha(q, x)$. In the case of a Moore automaton, the entry for the state q and the input x is the state $\delta(q, x)$ and a new column is added, whose entry for the state q corresponds to the output $\beta(q)$.

2.5.2 Transition Diagram

A trasition diagram is a directed graph, such that the set of verticles is the set of states Q of the automaton, and the directed edge from the state q to the state r exists if and only if there is a transition $\delta(q, x) = r$. In the case of a Mealy automaton, this edge is labeled by the pair $x/\alpha(q, x)$. In the case of a Moore automaton, the edge is labeled by symbol x and each vertex q is moreover labeled by $\beta(q)$.

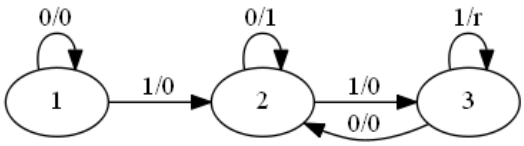
Example

Consider a Mealy automaton M, where Q= {1, 2, 3}, X = {0 ,1}, Y = {0, 1, r}, $\delta(1, 0) = 1$, $\delta(1, 1) = 2$, $\delta(2, 0) = 2$, $\delta(2, 1) = 3$, $\delta(3, 0) = 2$, $\delta(3, 1) = 2$, $\alpha(1, 0) = 0$, $\alpha(1, 1) = 0$, $\alpha(2, 0) = 1$, $\alpha(2, 1) = 0$, $\alpha(3, 0) = 0$ and $\alpha(3, 1) = r$. This automaton can be represented by a transition table and a transition diagram as follows:

Table 1: Example of a transition table

	0	1
1	1/0	2/0
2	2/1	3/0
3	2/0	3/r

Figure 1: Example of a transition diagram



2.6 Extented Transition and Extented Output Functions

2.6.1 Extented Transition Function

An extended transition function δ^* is the mapping $\delta^* : Q \times X^* \to Q$ defined inductively as follows:

1. $\delta^*(q, \epsilon) = q$ for all $q \in Q$
2. $\delta^*(q, vx) = \delta(\delta^*(q, v), x)$ for all $q \in Q, x \in X, v \in X^*$

Note that ϵ is the empty word.(See e.g. [2].)

2.6.2 Extented Output Functions

An extended output function of a Mealy automaton is the mapping

$$\alpha^+ : Q \times X^+ \to Y$$

satisfying the condition $\alpha^+(q, vx) = \alpha(\delta^*(q, v), x)$ for all $q \in Q, x \in X, v \in X^*$.

An extended output function of a Moore automaton is the mapping

$$\beta^* : Q \times X^* \to Y$$

satisfying the condition $\beta^*(q, v) = \beta(\delta^*(q, v))$ for all $q \in Q, v \in X^*$ (see [3], p. 58-59).

3 State of the Art

Although, the problem of congruences on finite-state machines is an important one, there are only a few papers that deal with it. In this section we will mention them.

In [3] basic facts about the automata theory are introduced, including definitions of an automaton state congruence, output automaton congruence, an intersection and an union of state congruences. Further, compositions and decompositions of automata through automaton state congruences are studied. In the last chapter there are given various algorithms for finite automata. One of them is an algorithm for finding minimal automaton state congruences, but not the elementary one, which is the topic of this thesis.

Papers [5], [6] and [7] also deal with congruences, but in a little different settings than we do. Because of it we have to bring in a few definitions and mention an important theorem for studying formal languages – the Myhill-Nerode theorem.

3.1 Definitions

A **finite-state transducer** is a sixtuple $T = (Q, X, Y, I, F, \delta)$ where Q, X, Y and δ have the same meaning as in **2.4.1**, I is a finite set of initial states ($I \subseteq Q$) and F is a finite set of output states ($F \subseteq Q$) (see [5]).

A **sequential finite-state transducer** is a transducer which has only one initial state.

A **language accepted by a transducer** is a set of all words $w \in L$ for which there exists $i \in I$ such that $\delta^*(i, w) \in F$ (see [2]).

Suppose that ϱ is an equivalence relation on a set X^*.

If for all $x, y, z \in X^*$ it holds that $x \,\varrho\, y$, then $xz \,\varrho\, yz$, equivalence ϱ is called a right congruence on X^*.

If a decomposion X^*/ϱ has a finite number of classes, then we say that ϱ has a finite index.

Myhill-Nerode theorem: A language $L \subseteq X^*$ is accepted by a transducer T if and only if a right congruence ϱ on X^* of a finite index exists such that L is a union of some classes of the right congruence ϱ (see [4]).

In the following paragraph, we will explore [5], [6] and [7] in more details.

In [5], various fields are first mentioned where finite-state automata are useful. The problem is that these automata can be very large so it is required to reduce the number of their states. Then it is noticed that the problem of a minimalization is solved for deterministic automata, because algorithms for computing reduced automata in an efficient way are known. The same does not hold for non-deterministic

sequential transducers. Then the algorithm *quasi-determination* is introduced that constructs an *intermediate transducer* which is deterministic. This algorithm affects only labels of a transducer, the number of states is not increased. Then the algorithm for a minimalization of sequential transcuders is introduced, which takes a intermediate transducer as an input. Further, its complexity is discussed and it is noticed that the algorithm can be also used for minimalization of p-sequential transducers, which are generalized sequential transducers.

[6] follows [5], there are discussed sequential transducers and their minimalization, too. The Myhill-Nerode theorem is generalized to sequential transduceres over unique GCD-monoid (it is mentioned that the Myhill-Nerode theorem also allows minimalization of transducers over other monoids and groups).

Let $\mathcal{A} = (A, \odot, \mathbf{1}, \mathbf{0})$ be a commutative monoid, where A is an algebraic structure, $\odot$ is an associative operation, $\mathbf{1}$ is the indentity element and $\mathbf{0}$ is the absorbing element. A monoid is called cancellation one if for all $a, a_1, a_2 \in A, a \neq \mathbf{0}$ it holds that if $a \odot a_1 = a \odot a_2$ and $a_1 \odot a = a_2 \odot a$, then $a_1 = a_2$. Let us quote parts of [5] to introduce result of this paper:

An *unique GCD-monoid* is a cancellation monoid $\mathcal{A} = (A, \odot, \mathbf{1}, \mathbf{0})$ in which 1) $a|\mathbf{1}$ implies $a = \mathbf{1}$, 2) a gcd (greatest common divisor) exists for every two non-zero elements, 3) a lcm (least common multiple) exists for every two nonzeroelements having a common multiple. In particular it yields that every gcd is indeed unique. We extend the definition of a gcd to arbitrary many elements as follows. Let $k \in \mathbb{N}_+$ and $\{a_1, \ldots, a_k\} \subseteq A \setminus \{\mathbf{0}\}$.

$$\operatorname*{gcd}_{\mathrm{i}\in\{1,\ldots,\mathrm{k}\}} a_i = \gcd(a_1, \gcd(a_2, \ldots, \gcd(a_{k-1}, a_k), \ldots))$$

with $\gcd_{i\in\{1\}} a_i = a_1$. Given an infinite set I and a family $(a_i)_{i\in I}$ we defined $\gcd_{i\in I} a_i = \gcd_{j\in J} a_j$ if there exists a finite set $J \subseteq I$ such that for every $i \in I$ there exists $j \in J$ with $a_j|a_i$. Otherwise, we define $\gcd_{i\in I} a_i = \mathbf{1}$.

Any mapping $S : X^* \to A$ is called *power series*. The set of all such power series is denoted by $A\langle\langle X^*\rangle\rangle$. We write (S, w) instead of $S(w)$ for all $S \in A\langle\langle X^*\rangle\rangle$ and $w \in X^*$. The support $\mathrm{supp}(S)$ of S is defined by $\mathrm{supp}(S) = \{w \in X^*, (S, w) \neq \mathbf{0}\}$.

For every $w \in X^*$ let $g(w) = \gcd_{u\in X^*, w.u\in \mathrm{supp}(S)}(S, w.u)$.

We call a power series $S \in A\langle\langle X^*\rangle\rangle$ directed, if for every $w \in \mathrm{supp}(S)$ we have $(S, w) = g(w)$.

Then it is proved that if a Myhill-Nerode congruence on a unique GCD-monoid has a finite index and power series are directed, then a minimal sequential transducer exists.

[7] deals with determinization of transducers over finite and infinite words. There is mentioned that determininazation of a transducer is a construction of a new transducer that defines the same function. The function in this text is a relation defined by transducers, it is a sequence of input and output symbols from an initial state to a finite state. Then sequential and subsequential functions and the right congruence over the set of input symbols are described, which allow to construct a minimal subsequential transducer realizing a function. Then the algorithm for finding determinization of transducers over finite words is introduced. The second part of the work deals with determinization of transducers over infinite words and

the algorithm which does so.

We can see that all [5], [6] and [7] are involved with finding the minimalization of transducers. In [5] an equivalence relation on X^* for minimalization is used, [6] and [7] follow the Myhill-Nerode theorem and its utilization for minimalization of automata.

In contrast with these papers, this thesis deals with a study of congruences defined over a set of states of automata and with finding elementary automaton state congruences which are the smallest automaton state congruence merging two distinct states. Notice that elementary congruences can be used for finding the minimal state automaton congruences described in [3].

4 Automaton State Congruences

4.1 Definition of an Automaton State Congruence

An automaton state congruence is an equivalence ϱ on the set Q of states of a given state automaton $M = (Q, X, \delta)$, such that for all states $q, r \in Q$ for which $q \,\varrho\, r$ and for all inputs $x \in X$ it holds that $\delta(q, x) \,\varrho\, \delta(r, x)$ (see [3], p. 66).[1]

4.2 Definition of an Output Automaton Congruence

An output automaton congruence is an automaton state congruence that satisfies that if $q \,\varrho\, r$, then $\alpha(q, x) = \alpha(r, x)$ for all $x \in X$ for a Mealy automaton, or $\beta(q) = \beta(r)$ for a Moore automaton (see [3], p. 66-67).

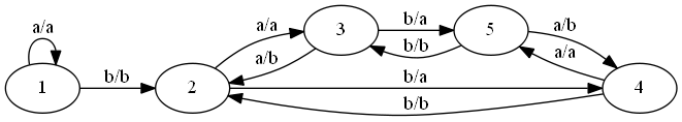
Thus, every output automaton congruence is an automaton state congruence, but on the other hand not every automaton state congruence is an output one.

Example

Let us consider a Moore automaton, whose transition diagram is displayed in the following figure.

The equivalence given by a decomposion $\{\{1\}, \{2, 4\}, \{3, 5\}\}$ is both an automaton state congruence and an output automaton congruence, and the equivalence given by a decomposion $\{\{1\}, \{2, 5\}, \{3, 4\}\}$ is only an automaton state congruence and not an output automaton congruence, because $\beta(2) \neq \beta(5)$ and $\beta(3) \neq \beta(4)$.

Figure 2: Example of automaton state and output automaton congruences



4.3 Intersection of Automata Congruences

4.3.1 Proposition

Given two state automaton congruences (output automaton congruences) ϱ and σ on M, then $\varrho \cap \sigma$ is a state automaton congruence (an output automaton congruence), too.

[1] In a general case, a congruence relation $\mathcal{R}$ is an equivalence relation on some algebraic stucture A with operations $\odot_1$, ..., $\odot_n$, which is compatible with all of these operations. Equivalent elements of the structure belong to the same equivalence (congruence) class.

4.3.2 Proof

The proof is straightforward, see e.g. [3].

4.3.3 Corollary

Given $\varrho_1, \ldots, \varrho_k$, $k \geq 1$, state automaton congruences. Then $\bigcap_{i=1}^{k} \varrho_i$ is a state automata congruence, too (see e.g. [3], p. 67).

4.4 Elementary Automaton State Congruence

Given a state automaton $M = (Q, X, \delta)$ and two distinct states $q, r \in Q$. The smallest state automaton congruence $\varrho_{q,r}$ for which $q \, \varrho_{q,r} \, r$ is called an elementary state automaton congruence ([3], p. 263).

4.4.1 Lemma

Let $M = (Q, X, \delta)$ be a state automaton and $q, r \in Q$ two distinct states. Then the elementary automaton state congruence $\varrho_{q,r}$ always exists.

4.4.2 Proof

Since ∇ is a state automaton congruence ($s \, \nabla \, t$ for all $s, t \in Q$), the collection of all state automaton congruences σ for which $q \sigma r$ is nonempty. The state automaton congruence $\varrho_{q,r}$ is now the intersection of all σ for which $q \, \sigma \, r$.

Corollary **4.3.3** finishes the proof.

Note that there is only a finite number of equivalence relations on a finite set Q.

4.4.3 Theorem

Let $M = (Q, X, \delta)$ be a state automaton and $q, r \in Q$ two distinct states, $(q \neq r)$. Then $s \, \varrho_{q,r} \, t$ if and only if either $s = t$, or there exist $a_1, \ldots, a_k \in X^*$ for $k \geq 1$ such that:

$$s \in \{\delta^*(q, a_1), \delta^*(r, a_1)\}, t \in \{\delta^*(q, a_k), \delta^*(r, a_k)\}$$

and for all $i = 1, \ldots, k-1$ it holds

$$|\{\delta^*(q, a_i), \delta^*(r, a_i)\} \cap \{\delta^*(q, a_{i+1}), \delta^*(r, a_{i+1})\}| \geq 1$$

4.4.4 Proof

Denote by σ the relation satisfying proposition in **4.4.3**. It holds that $q\,\sigma\,r$. Indeed, if we put $k = 1$ and $a_1 = \epsilon$, then $\delta^*(q, \epsilon) = q$ and $\delta^*(r, \epsilon) = r$.

We prove that σ is an equivalence relation on A, so we have to prove that the relation is reflexive, symmetric and transitive.

It is easy to see from the definition that σ is reflexive.

Let us prove symmetry: Take $s\,\sigma\,t$, $\{s, t\} \neq \{q, r\}$. Then there is a sequence $a_1, \ldots, a_k \in X^*, k \geq 1$ with the properties from the proposition. Now the sequence $a'_1 = a_k, a'_2 = a_{k-1}, \ldots, a'_k = a_1$ proves that $t\,\sigma\,s$.

We show that σ is transitive. Let $s\,\varrho_{q,r}\,t$ and $t\,\varrho_{q,r}\,u$. If $s = t$ or $t = u$, then we immediately get $s\,\varrho_{q,r}\,u$. Assume that $s \neq t$ and $t \neq u$. Then there are $a_1, \ldots, a_k \in X^*$ and $b_1, \ldots, b_l \in X^*$ such that

$$s \in \{\delta^*(q, a_1), \delta^*(r, a_1)\}, t \in \{\delta^*(q, a_k), \delta^*(r, a_k)\}$$

$$t \in \{\delta^*(q, b_1), \delta^*(r, b_1)\}, u \in \{\delta^*(q, b_l), \delta^*(r, b_l)\}$$

$$|\{\delta^*(q, a_i), \delta^*(r, a_i)\} \cap \{\delta^*(q, a_{i+1}), \delta^*(r, a_{i+1})\}| \geq 1$$

for $i = 1, \ldots, k-1$ and

$$|\{\delta^*(q, b_i), \delta^*(r, b_i)\} \cap \{\delta^*(q, b_{i+1}), \delta^*(r, b_{i+1})\}| \geq 1$$

for $i = 1, \ldots, l-1$.

Put $c_i = a_i$ for $i \leq k$, $c_i = b_{i-k}$ for $i = k+1, \ldots, k+l$. Then since $t \in \{\delta^*(q, c_k), \delta^*(r, c_k)\} \cap \{\delta^*(q, c_{k+1}), \delta^*(r, c_{k+1})\}$, the sequence $c_1, \ldots, c_{k+1}$ guarantees that the pair (s, u) is related by σ. That means $s\,\sigma\,u$.

Hence σ is an equivalence relation merging q and r.

Now, we have to prove that σ merges only those states that "have to be merged".

We prove that if $q\,\sigma_{q,r}\,r$, then for every $a \in X^*$ we have $\delta^*(q, a)\,\sigma_{q,r}\,\delta^*(r, a)$. We will prove this by induction on the length $|a|$ of a.

1. *Basic step*: If $|a| = 0$ or $|a| = 1$, then the assertion follows from the definition of a state automaton congruence.

2. *Inductive step*: Let us assume that the assertion holds for all $a \in X^*$, $|a| = k$. Take a with a length $|a| = k+1$, $a = x_1 \ldots x_{k+1}$. By the induction assumption we have:

$$s_1 = \delta^*(q, x_1 \ldots x_k)\,\sigma_{q,r}\,\delta^*(r, x_1 \ldots x_k) = t_1 \text{ (since } |x_1, \ldots, x_k| = k)$$

From the basic step, if $s_1\,\sigma_{q,r}\,t_1$, then $\delta(s_1, x_{k+1})\,\sigma_{q,r}\,\delta(t_1, x_{k+1})$ and

$$\delta(s_1, x_{k+1}) = \delta^*(q, x_1 \ldots x_{k+1});$$

$$\delta(t_1, x_{k+1}) = \delta^*(r, x_1 \ldots x_{k+1})$$

Now, since every equivalence relation is transitive, the proof is complete.

4.5 Factor State Automaton

Let us consider an automaton M = $(Q,\ X,\ Y,\ \delta,\ \alpha)$ and its automaton state congruence ϱ. Denote by $[q]_\varrho$ the equivalence class of ϱ which contains the state $q \in Q$.

A factor state automata $\bar{M}$ is defined as a quintuple $\bar{M} = (\bar{Q}, X, \bar{Y}, \bar{\delta}, \bar{\alpha})$ (see [3], p. 171), where:

- $\bar{Q} = Q/\varrho = \{[q]_\varrho;\ q \in Q\}$
- $\bar{Y} = \bar{Q} \times X$
- $\bar{\delta}([q]_\varrho, x) = [\delta(q, x)]_\varrho$
- $\bar{\alpha}([q]_\varrho, x) = ([q]_\varrho, x)$

Example

A Mealy automaton $M = (Q,\ X,\ Y,\ \delta,\ \alpha)$ is shown in the Figure 3. An equivalence ϱ such that 2 ϱ 5 and 3 ϱ 4 is an automaton state congruence. A factor state automaton $\bar{M} = (\bar{Q}, X, \bar{Y}, \bar{\delta}, \bar{\alpha})$ is defined as $\bar{Q} = \{[1] = \{1\},\ [2] = \{2,\ 5\},\ [3] = \{3,\ 4\}\}$, $X = \{a, b\}$, $\bar{Y} = \{([1], a), ([2], a), ([3], a), ([1], b), ([2], b), ([3], b)\}$ and it is displayed in the Figure 4.

Figure 3: Mealy automaton M

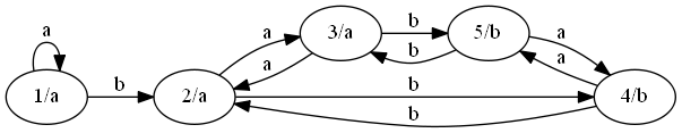
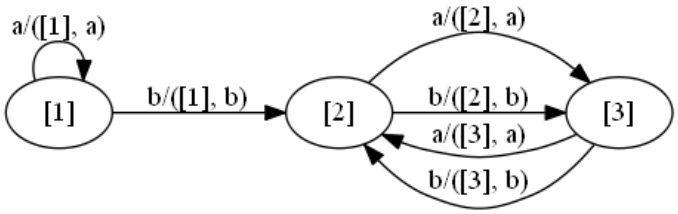


Figure 4: Factor state automaton $\bar{M}$



5 Construction of Elementary Automaton Congruences

Let us consider a state automaton $M = (Q,\ X,\ \delta)$ and a pair of states $q, r \in Q, q \neq r$. We will construct the elementary automaton congruence $\varrho_{q,r}$ merging these two states. During the work of the algorithm, $\mathcal{S}$ will denote a decomposion of the set Q.

At the beginning of the work, $\mathcal{S}$ contains one two-element set $\{q, r\}$, and all other sets are singletons.

During the work the following will be true: The decomposion to equivalence classes of $\varrho_{q,r}$ is "broader" then $\mathcal{S}$, i.e. for every set $A \in \mathcal{S}$ there is an equivalence class $[p]_\varrho$ such that $A \subseteq [p]_\varrho$.

A reason for making a union of two distinct sets $A, B \in \mathcal{S}$ is that there are $s, t \in C, C \in \mathcal{S}$ and $x \in X$ such that $\delta(s, x) \in A$ and $\delta(t, x) \in B$. That is why we will check all distinct pairs (s, t) in some set of $\mathcal{S}$, and if the sets A and B to which the transition function leads are not equal, we will make a union of A and B. It guarantees that at the end of the work of the algorithm, i. e. when there are no $s, t \in C$, $C \in \mathcal{S}$ and $x \in X$ such that $\delta(s, x) \in A, \delta(t, x) \in B$ for $A \neq B$, the set $\mathcal{S}$ will contain complete congruence classes.

We use different marks and pointers in the algorithm:

- $C \in \mathcal{S}$ is a working set;
- $(s, t) \in C$ is a working pair;
- n points to the next state within $A \in \mathcal{S}$;
- f points to the first state of the set $A \in \mathcal{S}$;
- l points to the last state of the set $A \in \mathcal{S}$;
- d points from some state to the "furtherst" processed state within $A \in \mathcal{S}$;
- *checked*(A) = *true* or *false* marks if all states of set $A \in \mathcal{S}$ were tested or not;
- *changedC* is set to *true* if the working set C is expanded and $s \neq f(C)$. The reason is explained in **5.3.1.4**.

5.1 Algorithm For Finding Elementary Automaton Congruences

Input: A state automaton $M = (Q,\ X,\ \delta)$ and a pair of states (q, r) such that $q \neq r$.

Output: The elementary automaton congruence $\varrho_{q,r}$ merging q and r given by its decomposion $\mathcal{S}$.

1. Initialize the set $\mathcal{S}$ as follows: $\mathcal{S} := \{\{q, r\}\} \cup \{\{p\}; p \in Q \setminus \{q, r\}\}$.
   Set $C := \{q, r\}$, $s := q$, $t := r$, $n(q) := r$, $f(C) := q$, $l(C) := r$, $checked(C)$ $:= false$ (for all one-element sets $A \in \mathcal{S}$ it holds that $checked(A) := true$), $changedC := false$;

2. For every $a \in X$:

   $S_1 := \text{find}(\delta(s, a))$, $S_2 := \text{find}(\delta(t, a))$;

   if $S_1 \neq S_2$:

   if $S_1 = C$ or $S_2 = C$:

   if $S_1 = C$:

   $A := S_2$;

   else

   $A := S_1$;

   end

   $C := C \cup A$; $n(l(C)) := f(A)$, $l(C) := l(A)$;

   if $|\mathcal{S}| = 1$

   go to step 6;

   end

   if $s \neq f(C)$

   $changedC = true$;

   end

   else

   $A := S_1 \cup S_2$; $n(l(S_1)) := f(S_2)$, $f(A) := f(S_1)$,

   $l(A) := l(S_2)$;

   Set a set A as unchecked ($checked(A) := false$);

   end

   end

   end for

   $d(s) := t$;

3. If $changedC = true$, then $s := f(C), t := n(d(s)), changedC = false$ and go to step 2;
   If $n(t)$ exists, then $t := n(t)$ and go to step 2;
   If $n(t)$ does not exist go to step 4;

4. if $n(s)$ exists and $n(s) \neq l(C)$:

   if $d(n(s))$ does not exist:

   $t := n(n(s))$, $s := n(s)$ and go to step 2;

   else

if $d(n(s)) \neq l(C)$

$t := n(d(n(s)))$, $s := n(s)$ and go to step 2;

end

end

end

5. Set $checked(C) := true$;
   If $S \in \mathcal{S}$ such that $checked(S) = false$ does not exist, go to step 6;
   Otherwise, $C := S$ and $s := f(C)$;
   If $d(s)$ does not exist (the set is used for the first time):

   $t := n(s)$ and go to step 2;

   else

   $t := n(d(s))$ and go to step 2;

   end

6. The algorithm finishes and returns the set $\mathcal{S}$.

5.2 Speeding Up of the Algorithm

The main disadvantage of the algorithm above is that we have to check all distinct pairs of states in each equivalence class. Consider the worst case: $\mathcal{S}$ contains two sets S_1 and S_2, $S_1 = Q \setminus \{p\}$, $S_2 = \{p\}$. So, during the work of the algorithm $\frac{(|Q|-1)(|Q|-2)}{2}$ pairs of states as a pair (s,t) were checked. ($|Q|$ is a size of set of states and $|Q| \geq 3$.) Luckily, there exists a possibility to speed the algorithm up.

Suppose that processing set $C \in \mathcal{S}$ contains three states u, v, w and it holds for some $x \in X$ that $\delta(u,x) \in U, \delta(v,x) \in V$ and $\delta(w,x) \in W$. In the first cycle of the algorithm $(s,t) := (u,v)$ and we unify sets U and V, $A_1 = U \cup V$. In the second cycle of the algorithm $(s,t) := (u,w)$ and we unify sets $A_2 = A_1 \cup W$, because $U \subset A_1$. It is clear that $A_2 = U \cup V \cup W$, so chosing (v,w) as a working pair becomes redundant. Of course, we would get the same result if we chose (v,w) instead of (u,w) as the first pair.

The conclusion is the following: It is not necessary to examine all distinct pairs in a set, we can hold the first state fixed and only move with t or skip some already checked block of states.

The accelerated algorithm is the same as the original one with a few differences: the first one is that it is enough to remember just the last processed element in some set, so d is not a function of a state but of a class now and we remember d(C) $= t$.

The next differences are that the step 4 is omitted and steps 3 and 5 looks as the following:

3a. if n(t) exists

if d(s) exists

$t = n(d(s))$ and go to step 2;

else

$t = n(t)$ and go to step 2;

end

end

5a. Set $checked(C) := true$;
If $S \in \mathcal{S}$ such that $checked(S) = false$ does not exist, go to step 6;
Otherwise, $C := S$ and $s := f(C)$;
if d(s) exists

$t = n(d(s))$ and go to step 2;

else

$t = n(s)$ and go to step 2;

end

Note that variable *changedC* is not used in the accelerated version.

5.3 Correctness of the Algorithm

We have to prove that the algorithm finishes its work and returns the correct output. We will consider the algorithm presented in **5.1**, the proof of the accelerated version will be similar.

5.3.1 Termination

In the next paragraph we will prove that the algorithm never suddenly finishes with some error, and never enters an endless loop.

5.3.1.1 Proposition

The algorithm finishes its work after a finite number of steps.

5.3.1.2 Proof

Q contains a finite number of states ($|Q|$ is a size of Q) and in the beginning of the algorithm $|Q| - 1$ sets are in $\mathcal{S}$. For some $x \in X$ there are two options in step 2: two sets of $\mathcal{S}$ are unified, so in $\mathcal{S}$ the number of pairs which have to be checked is increased, but the number of sets is decreased; or the number of pairs which have to be checked stays the same. In the worst case all sets except one are unified and the number of states, that we have to consider, is the biggest - because this number is finite and no pair of states is chosen twice or more times, the algorithm finishes after a finite number of steps. The same of course applies if the automaton can be decomposed to more than one equivalence class - because the number of equivalence

classes is finite and in each class there is a finite number of states and no pair of states is taken twice, the algorithm halts.

The fact that no pair is used more than once is proved in the next Lemma.

5.3.1.3 Lemma

If $\mathcal{S}$ is the system from the algorithm, then each pair (s,t) for which there is $A \in \mathcal{S}$ such that $s,t \in A$ will be checked exactly once.

5.3.1.4 Proof

In the original version of the algorithm it is necessary to make sure that each distinct pair of states is really checked, but to prevent the entering an endless loop at the same time. We do not have to check other pairs in the case of $\mathcal{S}$ contains only one set (that means $\varrho q, r$ is the universal equivalence class ∇), checking another pair (s,t) does not cause any change in $\mathcal{S}$, the algorithm halts and returns the correct output.

We divide the proof into two parts. At first we prove that for each distinct states $s,t \in A, A \in \mathcal{S}$ the pair (s,t) was checked. To be sure that no pair of states is omitted (of course except the case above when all states belong to the only equivalence class), it is important to keep a logical structure in the working set C. If the working set C is unified with another set, then s from the working pair (s,t) is set to the first element of the set C and t to the next unchecked state from s ($t := n(d(s))$). This assignment is correct, $n(d(s))$ has to exist because of the union of sets. This guarantees that, when we get to the end of the working set C, we can be sure that all distinct pairs of states in C were really taken and there are no unchecked "isles".

Secondly, we prove that no pair of states is checked more than once. Let us consider all possible situations which may occur during the work of the algorithm:

- *A working set C is used for the first time*:

  A working set C always contains two or more elements because it was either initialized in step 1, or some set A was assigned as C in step 5 (this set A is a result of an union of sets in step 2, so A contains two or more states).

  In both cases the first element of C is assigned as s and the second one as t ($t = n(s)$). Then s is fixed and in step 2 t is always "moved" to the next element until the end of set is reached – if it is so, then the condition that $n(t)$ does not exist and the algorithm goes to step 5 where a new s is chosen – this is described in the third item.

- *A working set C was already used*

  In this case a working set C contains one or more completely checked blocks of states. Because, when we make a union of a working set C and some set A in step 2, we put the set A behind the set C, one of the already checked blocks

is in the beginning of the set C. Hence we set $s := f(C)$ and $t := n(d(s))$. The reason for setting s as $f(C)$ is the same as above, $d(s)$ points to the last checked state from s, so $n(d(s)$ points to the next unchecked state. Note that $n(d(s))$ exists – it is the first element of the set A that was unified with the set C in step 2.

- *A working set C contains some already checked section*

  In this section we will discuss choosing of s in step 4. There are two possible situations (we consider that the first condition in step 4 is true, otherwise the whole step and choosing of s are skipped):

  If a state assigned as $n(s)$ was never chosen as s before, then $s = n(s)$ and $t = n(n(s))$ and the algorithm goes to step 2.

  If the state assigned as $n(s)$ was already chosen as s, then a whole block, which was already checked, exists. If $d(n(s))$ is not equal to the last element of C, then $(n(s), n(d(n(s))))$ has to be an unchecked pair of states. If $d(n(s)) = l(C)$, we can be sure that all pair of states from $n(s)$ to $l(C)$ were already checked. Indeed, this block from $n(s)$ to $l(C)$ had to be assigned as a working set C, so all pairs of these states were already checked.

We have proved that each distinct pair of states is checked at most once. Hence the algortihm always terminates.

The situation is of course easier in the accelerated version of the algorithm, because we do not require to check all distinct pairs of states and we simply remember the last checked state p in some set and new working pair is a pair $(p, n(p))$ (if $n(p)$ exists).

5.3.2 Correctness

5.3.2.1 Theorem

The algorithm constructs $\varrho_{q,r}$.

5.3.2.2 Proof

According to **4.4.3**, it suffices to prove that the decomposion $\mathcal{S}$ which is returned by the algorithm satisfies that $\delta^*(q, a), \delta^*(r, a)$ belong to the same set of $\mathcal{S}$ for all $a \in X^*$.
The work of the algorithm guarantees that for all $a \in X^*$ the states $\delta^*(q, a)$ and $\delta^*(r, a)$ will belong to i. e. the same $A \in \mathcal{S}$, to the same equivalence class of $\varrho_{q,r}$. We prove it by induction on a length $|a|$.

1. *Basic step*: If $a = \epsilon$ then $\delta^*(q, \epsilon) = q$ and $\delta^*(r, \epsilon) = r$, q and r belongs to the set C (initialisation).

2. *Inductive step*: Let $\delta^*(q,a)$ and $\delta^*(r,a)$ belong to the same set of $\mathcal{S}$ for all $a \in X^*$, $|a| = k$, $k \geq 1$. Let $b \in X^*$ have a length $|b| = k+1$. Then $b = ax$ for some $x \in X$.

   By the induction assumption, $s_1 = \delta^*(q,a)$ and $t_1 = \delta^*(r,a)$ belong to the same $S \in \mathcal{S}$. Since all pairs of S are checked, also s, t are checked for $x \in X$ and step 2 guarantees that $\delta^*(q,b) = \delta(s_1,x)$, $\delta^*(r,b) = \delta(t_1,x)$ will be in the same set of $\mathcal{S}$.

   The transitivity is guaranteed by the fact that we make a decomposion of $\mathcal{S}$. This completes the proof.

5.4 The Estimation of the Worst Processing Time of the Algorithm

5.4.1 Original Version

5.4.1.1 Proposition

The worst processing time of the algorithm is asymptotically equal to $O(|Q|^2|X|)$.

5.4.1.2 Proof

As it was said in **5.2**, the worst situation occurs when $\mathcal{S}$ contains two sets and one of them is one-element. In this case we have to check $\frac{(|Q|-1)(|Q|-2)}{2}$ pairs of states, so duration of the outer loop of the algorithm (until all distinct pairs are taken) is asymptotically equal to $O(|Q|^2)$. This number cannot be bigger, because no pair of states is chosen twice or even more on account of using pointers $d()$.

The inner loop of the algorithm takes place in step 2. This loop repeats itself for each $x \in X$ so it is asymptotically equal to $O(|X|)$, where $|X|$ is the size of the set X. All unions of sets take a constant amount of time because of using pointers n, f and l.

Steps 1, 3, 4, 5 and 6 also take a constant time because of using pointers.

In summary, the worst processing time of the algorithm is asymptotically equal to $O(|Q|^2|X|)$.

We assume that a finding of a set and an union of two sets take a constant time. More through discussion will be found in chapter **6**.

5.4.2 Accelerated Version

5.4.2.1 Theorem

The worst processing time of the accelerated version of the algorithm is asymptotically equal to $O(|Q||X|)$.

5.4.2.2 Proof

The omitting of the step 4 in the accelerated version of the algorithm does not cause any change in the worst case analogous estimation, because step 4 has a constant duration.

The same holds for a modification of function of the pointer d - there is no change of the estimation again.

The estimation of duration of the outer loop is of course different. The number of taken pairs of states decreases to $|Q| - 1$ in the worst case, so the duration is asymptotically equal to $O(|Q|)$.

The duration of the accelerated version is asymptotically equal to $O(|Q||X|)$ overall.

6 Implementation of the Algorithm

In this section we will focus on implementation of both versions of the algorithm described in the previous chapter. There we considered sets as something "abstract" and we did not handle how to represent them in a computer. However, it is imporant to find a way how to do so. Luckily, an algorithm exists which solves this problem.

The algorithm from chapter **5.1** was implemented in Matlab version R2014a using the CTU licence for students.

6.1 Representing of the Set Structure and Initialization

Let $M = (Q, X, \delta)$ be a state automaton, $|X| = l$, $|Q| = m$.

Let us assume that states are assigned by numbers from 1 to m. It can always be done without loss of generality because two automata will be equivalent if they differ only in names of states.

Let δ be represented by an array *delta* on the size $m \times l$ where transitions are stored. So, if for $a, b \in Q, x \in X$ it holds that $\delta(a, x) = b$, it is represented as $delta(a, x) = b$.

Let us assume that all sets $S \in \mathcal{S}$ are represented by its first element. It helps us to hold information which set is checked and which is not by creation of an array *checked* for which it holds: if $a \in Q$ is the first element of some unchecked set $S \in \mathcal{S}$, then $checked(a) = 0$ and for all elements $b \in S, b \neq a$ it holds $checked(b) = 1$. Of course, if a set $S \in \mathcal{S}$ is completely checked, then for all elements $a \in S$ it holds that $checked(a) = 1$. Let us remind that, in the initilazation of the algorithm, $\mathcal{S}$ contains one two-element set $\{q, r\}$ and other sets are singletons. So *checked* is an array of ones, the only exception is that $checked(q) = 0$.

To implement step 2 in the algorithm, we have to find a way to easily get a set to which some element belongs (so-called operation *find*). So, we need a representant of every set to which all elements from some set will point to. Because we already use the first element of a set as its representant, we do it also in this case. Now it is very straightforward to use the union-find algorithm described in [8] and [9] using two heuristic for improving the processing time, in [8] they are assigned as the *collapsing rule* and the *weighted union rule*, in ([9], p. 569) as the *path compression* and the *union by rank*.

In the accelerated version of the algorithm we can use both of these heuristic, but in the original version if we make an union of a working set C with another set A we are not able to use the *weighted union rule*. When we use *weighted union rule* after an union of two sets the smaller set is a successor of the bigger one. In our case a working set C has to always be an ancestor of a set A to keep the structure of C.

Changes appear in the case of the accelerated version too. Because we use the *weighted union rule*, it can easily happen that the working set C gets behind another (bigger) set. In this case, we have to the first element of the unified set assigned as s to keep the information which pairs of states were checked and which were not.

In our description of the algorithm we assumed that operations *find* and *union* take a constant time. Of course, it does not hold anymore when we use the union-find algorithm. Let us denote by k a number of *union* operations and by m a number of *find* operations. According to [10], if $m \geq k$ and we use *union by rank* and *path compression*, then the worst processing time is $\Theta(m\,\alpha(m,k))$, where α is the inverse Ackermann function, when we use naive linking (union) and *path compression*, then the worst processing time is $\Theta(m\,\log k)$. If $m \leq k$, and we use both *union by rank* and *path compression*, then the worst processing time is $\Theta(k + m\,\alpha(k,k))$, when we use only a *path compression*, then the worst processing time is $\Theta(k + m\,\log k)$. Note that in our case the number of operations *find* is always bigger than the number of *union* operations.

Let us go back to describing the implementation. We need only to remember some "set index" (this means the first element of a set which is assigned as C) to represent the working set C.

Implementation of pointers n and d is very straightforward, both are implemented as arrays of size $1 \times m$. The same holds for the pointer l. Let us assume that $a \in S, S \in \mathcal{S}$ is the first element of S. If $l(a) = b$, it means that b is the last element of set S.

6.2 Code

The attached CD contains the following:

Data that we used for testing the algorithm, they include data sets we will describe in the next chapter. They are called: *10-60.mat*, *10-60_ns.mat*, *100-1000.mat*, *100-1000_ns.mat*, *10000-100000.mat*, *trans_by_5.mat*, *trans_by_10.mat*, *30trans.mat*. Note that these data sets contain arrays with transitions.

main.m is the script, from which the program is launched, and where an array *delta* is loaded from data files, and a pair (q, r) is defined.

Functions which are used for finding the elementary congruence:

- *elementary_congruence.m*:

  Input: an array *delta*, a array *qr* of size 1×2 containg a pair (q, r), *ver* = 1 (the original version) / 2 (the accelerated version)

  Output: array *output* of size $1 \times m$ containing for all elements a number to which equivalence class belong

  It is the main function, body of the program, from which all another functions are called, structure of this function matches the structure of the algorithm represented in **5.1**.

- *init.m*:

  Input: *delta*, *qr*

  Output: A structure *output* of the array *st* of the size 1×2 which contains states s and t, *c_index* which is the number of the set C and the

array *init_arrays* which is two dimensional array of the size $5 \times m$, where rows correspond to the arrays *checked*, $first\_element$, n, l and d.

In this function structures are initialized that are used during the work of the algorithm. This function is called in the beginning of the fuction *elementary_congruence.m*.

- *finding.m*:

  Input: array x of elements $S1$ and $S2$ (see in **5.1**), arrays $first\_element$ and l

  Output: array o of are the first elements of the set to which $S1$ and $S2$ belong, modificated arrays $first\_element$ and l

  Function for finding the representant of sets. There is modificated an array $first\_element$ and l too to holding the information to which class some element belongs.

- *union.m*:

  Input: numbers a, b which represent the set we want to make a union of, $first\_element$, n, l, *checked*, *c_index*, optional argument which it is used only if elements are added into the working set C in the original version.

  Output: changed arrays $first\_element$, n, l, *checked*, *c_index*

  Function which makes a union of two sets. The reason, why *c_index* is one of input arguments a output ones too, is the following: When we use the accelerated version and we make a union of a working set C and some bigger set, we have to s assign as the first element of the bigger set to get correct output.

- *get_st*:

  Input: *st*, *c_index*, $first\_element$, n, l, d, *ver* and the optional argument which is used only if elements are added into the working set C in the original version.

  Output: changed *st*

  Finds new (s, t) according to the conditions of one of individual versions. If new (s, t) is not found, either $st(1)$ or $st(2)$ contains a zero which it is used as a condition for the entering function *get_C*.

- *get_C*:

  Input: *checked*, *st*, *c_index*, $first\_element$, n, d, *ver*

  Output: changed *checked*, *st* and *c_index*

  Returns an index of a unchecked set a initializes new (s, t). If all sets are checked, then program halts (after a condition in *elementary_congruence*).

- *write_classes*:

  Intput: $first\_element$, Output: *classes*

  Returns all states of a automaton decomposed to the equivalence classes.

7 Experiments

In this section we will discuss the differences of both versions of the algorithm due to the real processing time. The program was launched on a processor Intel Core i3, the results will be of course different when we will use another processor, but it suffices to have an idea if the accelerated version is really faster than the original one (as we suppose) and to find the limits of the algorithm.

Both versions of the algorithm run under the same conditions on the same data set to get the best comparison of them.

7.1 Data

In this paragraph we will introduce the data we used for testing. We had available two kinds of automata: The first group let us call as *trivial automata*, the automata which have only two congruences – the indentical and the universal congrueneces. The second group we will call *nontrivial automata*. These are the automata which have some nontrivial congruence. The reason, why we chose to testing both groups, is the following: States of trivial automata will belong to the only congruence class in the end of the running of the algorithm, so we expect that durations of both version of the algorithm will be almost the same. On the other hand, if a nontrivial congruence exists, we expect that the accelerated version will be faster than the original one.

For testing we used automata of various sizes, the list of them follows:

- five trivial automata, five nontrivial automata of the size $|Q| = 10, \ldots, 60$, $|X| = 2$;
- two trivial automata, two nontrivial automata of the size $|Q| = 100, \ldots, 1000$, $|X| = 2$;
- one trivial automaton of the size $|Q| = 100, \ldots, 1000$, $|X| = 5.(|Q|/100)$;
- one trivial automaton of the size $|Q| = 100, \ldots, 1000$, $|X| = |Q|/10$;
- one trivial automaton of the size $|Q| = 100, \ldots, 1000$, $|X| = 30$;
- one trivial automaton of the size $|Q| = 10\ 000$, $|X| = 2$;
- one trivial automaton of the size $|Q| = 100\ 000$, $|X| = 2$;

The processing time of the algorithm on a automaton also depends on choosing (q, r). Because of it we launched the algorithm on small automata (ten to sixty states) with all distinct combinations of (q, r) and then made an average of given processing times. Each of these settings we launched ten times, because processing times were small and results could be changed just because of the running of the

computer.

On bigger automata we did not use all combinations of (q, r) (e.g. for 100 states we will have to run the program for almost five thousand of pairs (q, r)), we randomly chose a few distinct pairs (q, r), then we run algortihms on them and also made an average of given durations.

When there were more than one automata of a category (e.g. five trivial automata of ten states), we made an average of their processing times, when there was only one automaton of one setting, we launched the algorithm on this automaton more times and then made an average of these durations.

7.2 Results

Note that in this paragraph we will write OA when we used the original algorithm and AA when we used the accelerated version. Trivial automata are assigned as T, nontrivial automata as N.

7.2.1 Automata of 10-60 States

We used for testing trivial and nontrivial automata with number of states from ten to sixty and with two input symbols. The durations of both versions of the algorithm you can find in **Appendix** in the **Table 2**. In the **Figure 6**, times of the original version running on trivial and nontrivial automata are compared and you can see the same for the accelerated version in the **Figure 7**. Durations between the original and the accelerated version running on trivial automata are plotted in the **Figure 8**, running on nontrivial automata is displayed in the **Figure 9**.

We can see that the difference of durations of both versions on trivial automata is minimal, the accelerated version is faster on nontrivial automata than the original version, and moreover the duration of accelerated version on both groups of automata is almost the same.

We can see on graphs that all running times resemble linear.

7.2.2 Automata of 100-1000 States

The situation here is almost the same as in **7.2.1**. We can see measured values in **Appendix**, in the **Table 3**. We split the graph of durations of the original algorithm on trivial and nontrivial automata to two graphs **Figure 10** and **Figure 11** to preserve legibility. In the **Figure 12** we can see comparison of times of the accelerated version of the algorithm on trivial and nontrivial automata. We see comparison of original and accelerated version on trivial automata in the **Figure 13** and on nontrivial automata in the **Figure 14**.

Results are similar as before. The accelerated version runs almost equally on trivial and nontrivial automata, the original version is slower on nontrivial automata than on trivial ones. The running time of the original version on nontrivial automata seems to be quadratic, this responds to the proposition we made in **5.4.1.1**.

7.2.3 Automata With More Than Two Input Symbols

The next question is if various size of $|X|$ affect the processing time of the

algorithm. We will use only trivial automata. In the previous paragraph, we saw that if $|X| = 2$, then processing times of both versions of the algorithm are almost the same. We would like to figure out if a bigger size of $|X|$ will cause any changes.

Versions of the algorithm are launched on automata that have from hundred to thousand states. In the first data set, the size of $|X|$ is incremented by five (so if an automaton has hundred states, $|X| = 5$, if an automaton has two hundred states, $|X| = 10$, etc.), in the second data set the size of X is incremented by ten (if $|Q| = 100$, $|X| = 10$, if $|Q| = 200$, $|X| = 20$, etc.) and the size of $|X|$ is always 30 in the third data set.

The measured values are in the **Table 4** for the original version and in the **Table 5** for the accelerated version. In the figures **Figure 15** (the original version) and **Figure 16** (the accelerated version) these values are plotted. We can see that all values are practically the same. The same result we get if we compare original and accelerated version (**Figure 17**: $|X|$ is incremented by five, **Figure 18**: $|X|$ is incremented by ten, **Figure 19**: $|X| = 30$).

The results are interesting, because we expected that the more input symbols are the faster all states get into one class. It is probably caused by the structure of tested automata, where many input symbols point to states that was already in C.

7.2.4 Automata With Big Number of States

We compared both versions of the algorithm on big data sets – on nontrivial automata with ten thousand and hundred thousand states. You can find measured values in the **Table 6** in **Appendix**.

It is clear why chosing of the accelerated version of the algorithm is more preferable. The duration of the accelerated version on one hundred thousand states took about ten minutes, the original version took couple hours already on ten thousand of states.

7.2.5 Conclusion

Let us summerize what we obtained in this section. It does not matter whether we will use the original or the accelerated version of the algorithm on trivial automata, the accelerated version is only slightly faster. When we examine nontrivial automata, the best choice is choosing the accelerated version. Because in practice, we often cannot know if an automaton is trivial or nontrivial, we recommend to always use the accelerated version.

We also saw that size of the set $|X|$ does not affect the processing time of the algorithm on trivial automata. As we mentioned in **7.2.3**, it is probably caused by the structure of examined automata.

8 The Smallest Automaton State Congruence Containing a Relation

In the fifth chapter we introduced an algorithm for finding elementary automaton state congruences which takes a state automaton $M = (Q, X, \delta)$ and two distinct states $q, r \in Q$, $q \neq r$ as an input and returns the elementary automaton state congruence that merges these two states q and r.

Let us consider a slightly different problem: Let be $M = (Q, X, \delta)$ a state automaton and R a relation on Q. We would like to construct the smallest automaton state congruence ϱ that contains R. This problem can be solved by a modification of the algorithm from chapter **5**, we will use the accelerated version.

1. Construct a set $\mathcal{C}$ of connected components of the graph $G = (V, R)$.
   Put $\mathcal{S} := \mathcal{C}$.
   For all $S \in \mathcal{S}$, $S = \{a_1, \ldots, a_k\}$, $|S| \geq 2$:
   For all $i = 1, \ldots, k-1$;
   $n(a_i) := a_{i+1}$;
   end
   end
   Choose $S \in \mathcal{S}$, $|S| \geq 2$, put $C := S$.
   checked$(C) :=$ *false*, $s := f(C)$, $t := n(s)$.
   For all $A \in \mathcal{S}, A \neq C, |A| \geq 2$ set *checked*$(A) :=$ *false*.

2. For every $a \in X$:
   $S_1 := \text{find}(\delta(s, a))$, $S_2 := \text{find}(\delta(t, a))$;
   if $S_1 \neq S_2$:
   if $S_1 = C$ or $S_2 = C$:
   if $S_1 = C$:
   $A := S_2$;
   else
   $A := S_1$;
   end
   $C := C \cup A$; $n(l(C)) := f(A)$, $l(C) := l(A)$;
   if $|\mathcal{S}| = 1$
   go to step 5;
   end
   else
   $A := S_1 \cup S_2$; $n(l(S_1)) := f(S_2)$, $f(A) := f(S_1)$,
   $l(A) := l(S_2)$;

checked$(A) :=$ *false*;

end

end

end for

$d(C) := t$;

3. if n(t) exists

if d(s) exists

$t = n(d(s))$ and go to step 2;

else

$t = n(t)$ and go to step 2;

end

end

4. Set *checked*$(C) :=$ *true*;
If $S \in \mathcal{S}$ such that *checked*$(S) =$ *false* does not exist, go to step 5;
Otherwise, $C := S$ and $s := f(C)$;
if d(s) exists

$t = n(d(s))$ and go to step 2;

else

$t = n(s)$ and go to step 2;

end

5. The algorithm finishes and returns the set $\mathcal{S}$.

Proof of the correctness of this algorithm is very similar to the one from **5.3** and we omit it. The same holds for the estimation of the worst processing time.

Note that the algorithm described in **5.1** is the special case of this algorithm where $R = \{(q, r)\}$.

Example

Consider a state automaton $M = (Q, X, \delta)$, $Q = \{1, 2, 3, 4, 5, 6, 7\}$, $X = \{a, b\}$ which is displayed in the **Figure 5** and a relation R which is defined as: for $p, o \in Q$ it holds that pRo when $\delta(p, b) = \delta(o, b)$. We can see that the relation R decomposed states of the automaton to the equivalence classes: $[1]_R = \{1, 2, 5\}$, $[2]_R = \{3, 4\}$, $[6]_R = \{6, 7\}$.

When we prepare an input as it was described above and launch the algorithm on it, we will get $\mathcal{S} = \{\{1, 2, 5\}, \{3, 4, 6, 7\}\}$, which is the smallest automaton state congruence ϱ containing the relation R. A factor state automaton $\bar{M} = (\bar{Q}, X, \bar{\delta})$, where $\bar{Q} = \{[1]_\varrho = \{1, 2, 5\}, [3]_\varrho = \{3, 4, 6, 7\}\}$, corresponds to the state automaton M. We see that the smallest automaton state congruence contains R.

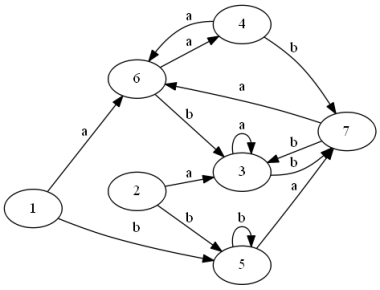


Figure 5: State automaton M

9 Conclusion

Let us summerize the content of this thesis. We have introduced an elementary automaton state congruence and we have proved that this congruence always exists. Then we brought in two versions of an algorithm for finding this congruence, proved its correctness, estimated the worst processing time, and then describted its implementation. We discussed results on various data sets and proved that it is more advantageous to use the accelerated version than the original one. We also introduced known results in the area of studying automata congruences.

We believe that presented algorithm will be interesting for people involved in the study of finite automata and will be helpful in practice. Finding of minimal automaton state congruences is other not fully discovered field. In [3] the algorithm for finding the minimal automaton state congruence is introduced, it will be interesting to implement it and also discuss the results on examples. The algorithm for finding the elementary automaton state congruence can help in constructing of the minimal ones.

References

1: Hopcroft, J. E., Motwani R., Ullman J. D. (2001) *Introduction to Automata Theory, Languages, and Computation*, second edition, Addison-Wesley, Chapter 1: The Methods and the Madness, p. 1-36,
ISBN: 0-201-44124-1

2: Demlová M. (2014) Lectures to *Jazyky, automaty, gramatiky* on CTU, FEE
URL: *http://math.feld.cvut.cz/demlova/teaching/jag/predn_jag.html*

3: Demlová M., Koubek V. (1990) *Algebraická teorie automatů*, Nakladatelství technické literatury
IBSN: 80-03-00348-2

4: Kozen D. (2008) Lectures to *Theory of Computation* on Cornell University, FCCS
URL: *http://www.cs.cornell.edu/Courses/CS682/2008sp/Handouts/MN.pdf*

5: Mohri M. (1996) Minimalization algorithms for sequential transducers, AT&T Laboratories, 177-201
URL *http://www.sciencedirect.com/science/article/pii/S0304397598001157*

6: Malleti A. Myhill-Nerode Theorem for Sequential Transducers over Unique GCD-Monoid, Dresden University of Technology, D-1062
URL: *http://www.ims.uni-stuttgart.de/institut/mitarbeiter/maletti/pub/mal04b-full.pdf*

7: Béal M., Carton O. (2001) .Determinization of transducers over finite and infinite words, Universite de Marne-la-Vall, 255-251
URL: *http://www.sciencedirect.com/science/article/pii/S0304397501002717*

8: Tarjan, R. E. (1975) Efficiency of a Good But Not Linear Set Union Algorithm, Journal of the ACM (JACM), p. 215-225
URL: *http://dl.acm.org/citation.cfm?id=321884*

9: Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2009) *Introduction to Algorithms*, third edition, MIT Press and McGraw-Hill, Chapter 21: Data Structures of Disjoint Sets, p. 561-585
IBSN: 978-0-262-03384-8

10: Tarjan, R. E., van Leeuwen, J. (1984) Worst-case Analysis of Set Union Algorithms, Journal of the ACM (JACM), p. 245-281
URL: *http://dl.acm.org/citation.cfm?id=2160*

10 Appendix

A Tables

Table 2: Automata of 10-60 states

All values are in seconds.

	10 states	20 states	30 states	40 states	50 states	60 states
OA, T	0.00158	0.00238	0.00444	0.00524	0.00615	0.00784
OA, N	0.00285	0.01257	0.02384	0.05395	0.06205	0.07391
AA, T	0.00110	0.00205	0.00417	0.00511	0.00604	0.00731
AA, N	0.00102	0.00227	0.00319	0.00521	0.00584	0.00671

Table 3: Automata of 100-1000 states

All values are in seconds.

	100 states	200 states	300 states	400 states	500 states
OA, T	0.01189	0.02396	0.03855	0.04779	0.06552
OA, N	0.21344	1.01214	2.22224	2.12883	6.54510
AA, T	0.01261	0.02334	0.04149	0.04315	0.06155
AA, N	0.00940	0.02063	0.03241	0.03806	0.05362

	600 states	700 states	800 states	900 states	1000 states
OA, T	0.07724	0.09685	0.12572	0.15453	0.17232
OA, N	9.68674	12.55752	18.17188	22.35938	30.64867
AA, T	0.07428	0.09428	0.11896	0.14456	0.16235
AA, N	0.07295	0.08046	0.09848	0.11126	0.12941

Table 4: Automata with more input symbols, the original version

All values are in seconds.

	100 states	200 states	300 states	400 states	500 states
$\|X\| = 5.(\|Q\|/100)$	0.01372	0.03272	0.04717	0.07634	0.09451
$\|X\| = \|Q\|/10$	0.01161	0.02448	0.04538	0.06535	0.07798
$X = 30$	0.00966	0.03423	0.05379	0.06167	0.07477
	600 states	700 states	800 states	900 states	1000 states
$\|X\| = 5.(\|Q\|/100)$	0.12046	0.14077	0.19933	0.20956	0.23842
$\|X\| = \|Q\|/10$	0.11834	0.15942	0.15486	0.16746	0.18951
$X = 30$	0.139284	0.13172	0.20680	0.21184	0.23609

Table 5: Automata with more input symbols, the accelerated version

All values are in seconds.

	100 states	200 states	300 states	400 states	500 states
$\|X\| = 5.(\|Q\|/100)$	0.01390	0.03541	0.04647	0.08994	0.09818
$\|X\| = \|Q\|/10$	0.01103	0.03018	0.04660	0.06629	0.06958
$X = 30$	0.01096	0.03232	0.05430	0.05888	0.07694
	600 states	700 states	800 states	900 states	1000 states
$\|X\| = 5.(\|Q\|/100)$	0.12250	0.14068	0.16766	0.20435	0.27684
$\|X\| = \|Q\|/10$	0.10687	0.16071	0.16177	0.17576	0.19328
$X = 30$	0.14598	0.12277	0.20513	0.22235	0.23175

Table 6: Automata of 10 000 and 100 000 states

All values are in seconds.

	10 000 states	100 000 states
OA	23 075.705	–
AA	4.696	604.461

B Graphs

Figure 6: Original version running on trivial and nontrivial automata of 10-60 states

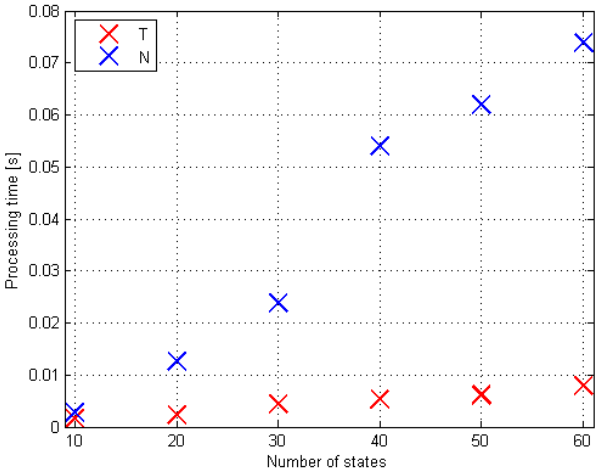


Figure 7: Accelerated Version running on trivial and nontrivial automata of 10-60 states

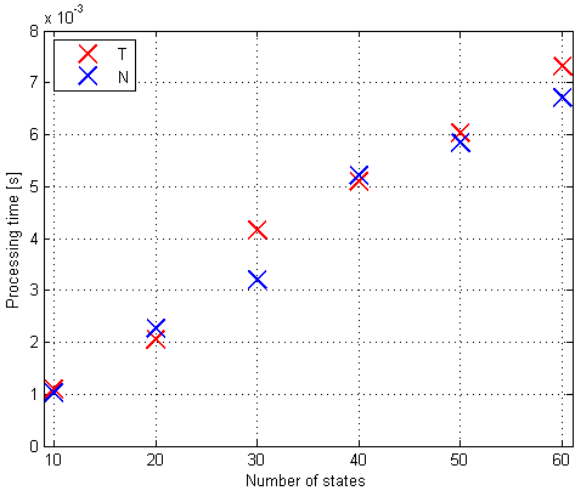


Figure 8: Comparison of OA and AA running on trivial automata of 10-60 states

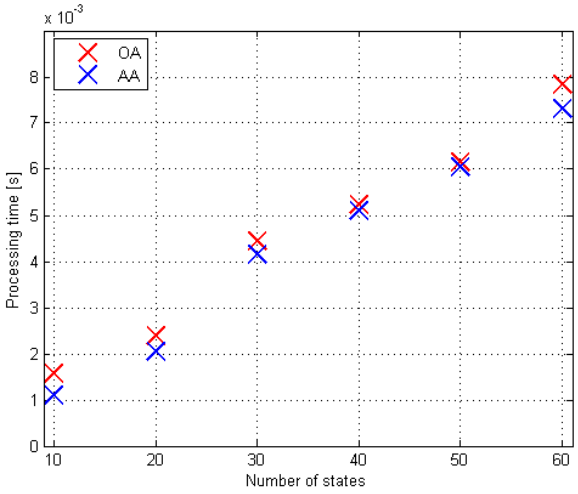


Figure 9: Comparison of OA and AA running on nontrivial automata of 10-60 states

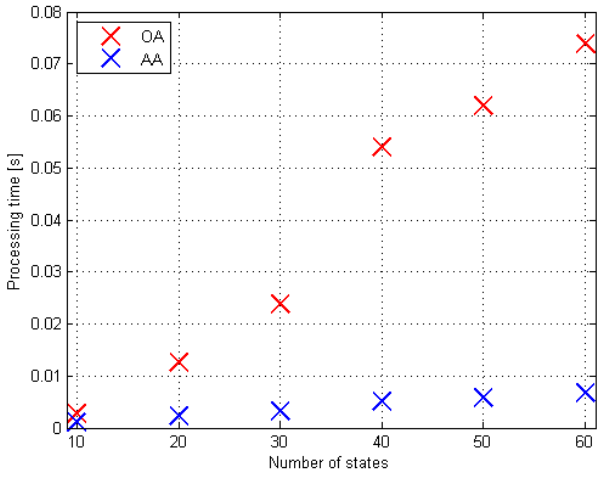


Figure 10: The original version running on trivial automata of 100-1000 states

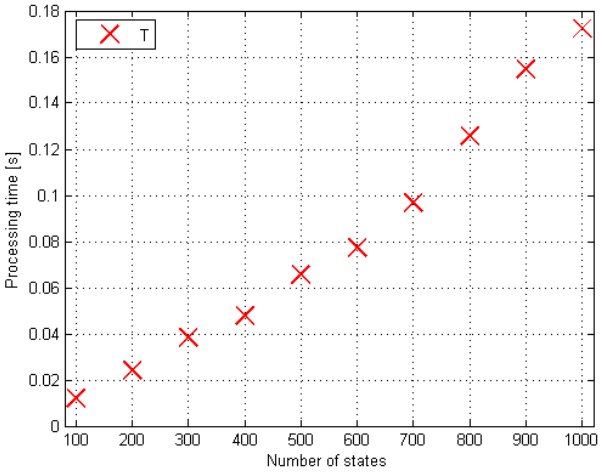


Figure 11: Original version running on nontrivial automata of 100-1000 states

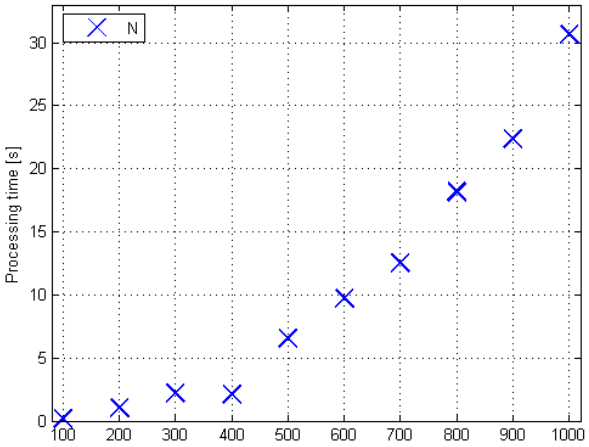


Figure 12: Accelerated version running on trivial and nontrivial automata of 100-1000 states

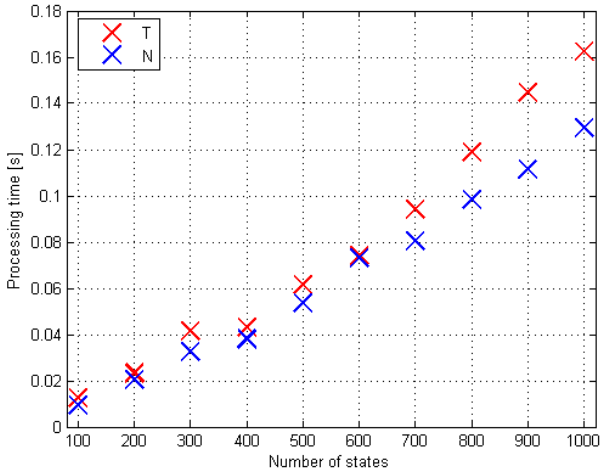


Figure 13: Comparison of OA and AA running on trivial automata of 100-1000 states

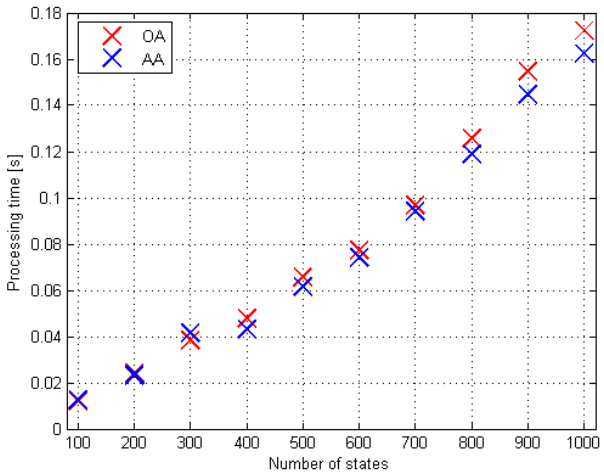


Figure 14: Comparison of OA and AA running on nontrivial automata of 100-1000 states

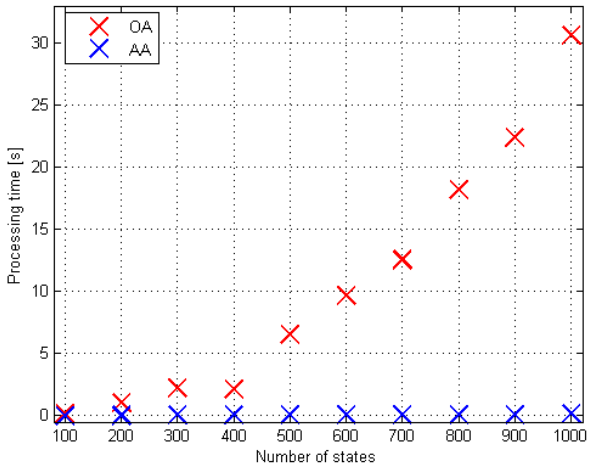


Figure 15: Original version running on trivial automata with more input symbols

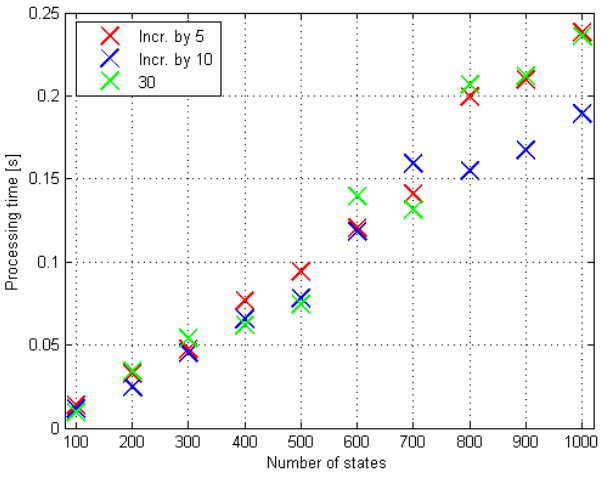


Figure 16: Accelerated version running on trivial automata with more input symbols

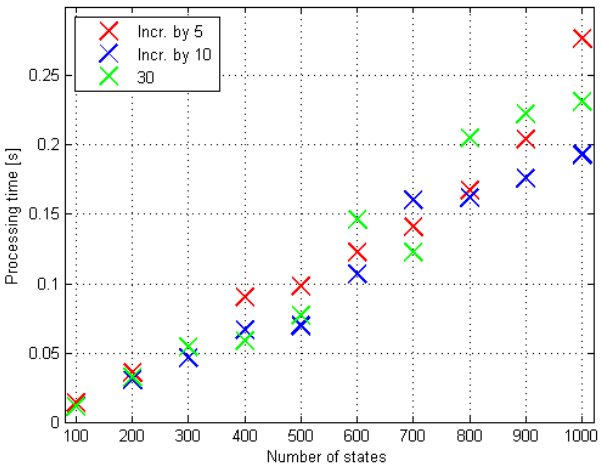


Figure 17: Comparison of OA and AA running on trivial automata with input symbols incremented by 5

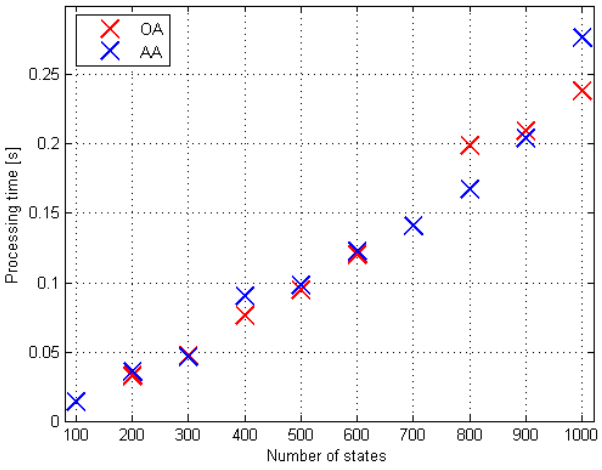


Figure 18: Comparison of OA and AA running on trivial automata with input symbols incremented by 10

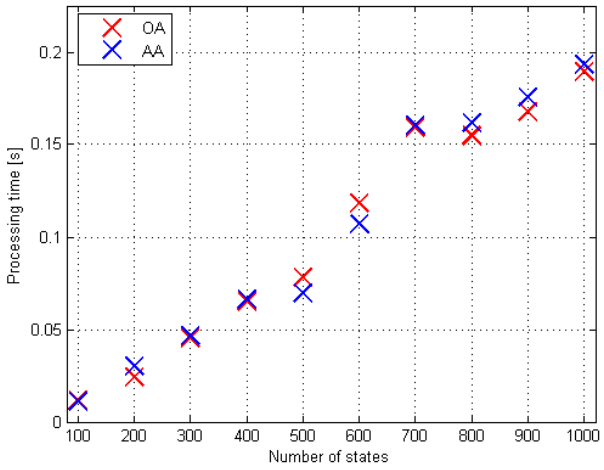
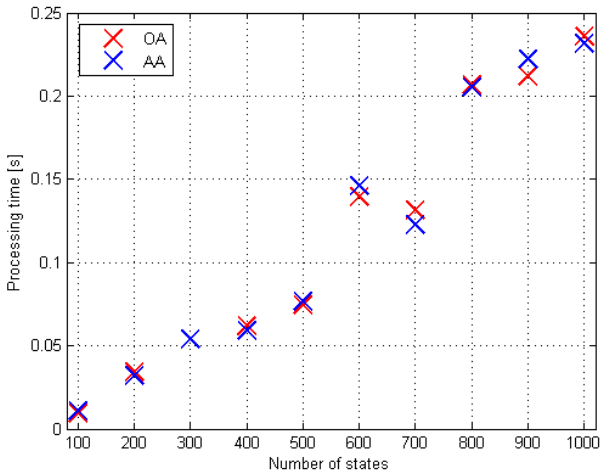


Figure 19: Comparison of OA and AA running on trivial automata with input symbols equal to 30



C Contents of CD

Attached CD contains the following:

- Electroning version of this thesis in PDF format
- Folder called *program* containing the source code describted in **6.2**