Lecture Notes in Computer Science 2214
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Oliver Boldt   Helmut Jürgensen (Eds.)

Automata Implementation

4th International Workshop on
Implementing Automata, WIA'99
Potsdam, Germany, July 17-19, 1999
Revised Papers

Springer

Volume Editors

Oliver Boldt
Universität Potsdam, Institut für Informatik
August-Bebel-Straße 89, 14482 Potsdam, Germany
E-mail: boldt@cs.uni-potsdam.de

Helmut Jürgensen
Universität Potsdam, Institut für Informatik
and
The University of Western Ontario, Department of Computer Science
London, Ontario, Canada N6A 5B7
E-mail: helmut@uwo.ca

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Automata implementation : revised papers / 4th International Workshop on Implementing Automata, WIA ’99, Potsdam, Germany, July 17 - 19, 1999. Oliver Boldt ; Helmut Jürgensen (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2214)
ISBN 3-540-42812-7

CR Subject Classification (1998): F.1.1, F.4.3, I.2.7, I.2.3, I.5, B.7.1

ISSN 0302-9743
ISBN 3-540-42812-7 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10840850 06/3142 5 4 3 2 1 0

Foreword

This volume contains the revised versions of papers presented at the fourth international Workshop on Implementing Automata (WIA), held 17–19 July, 1999, at Potsdam University, Germany.

As for its predecessors, the theme of WIA99 was the implementation of automata and grammars of all types and their application in other fields. The papers contributed to this volume address, among others, algorithmic issues regarding automata, image and dictionary storage by automata, and natural language processing.

In addition to the papers presented in these proceedings, the workshop included a paper on quantum computing by C. Calude, E. Calude, and K. Svozil (published elsewhere), an invited lecture by W. Thomas on *Algorithmic Problems in the Theory of ω-Automata,* a tutorial by M. Silberztein on the INTEX linguistic development environment, and several demonstrations of systems.

The local arrangements for WIA99 were conducted by Helmut Jürgensen, Suna Aydin, Oliver Boldt, Carsten Haustein, Beatrice Mix, and Lynda Robbins. The meeting was held in the *Communs* building, now the main university building, of the New Palace in the park of Sanssouci, Potsdam.

The program committee for WIA99 was:

A. Brüggemann-Klein	Technische Universität München
J.-M. Champarnaud	Université de Rouen
F. Günthner	Universität München
H. Jürgensen	Universität Potsdam and University of Western Ontario
D. Maurel	Université de Tours
D. Raymond	Gateway Group Inc.
K. Salomaa	University of Western Ontario
W. Thomas	Rheinisch-Westfälische Technische Hochschule Aachen
B. Watson	Ribbit Software Systems Inc.
D. Wood	Hong Kong University of Science and Technology
S. Yu	University of Western Ontario

The work of the program committee, the reviewers, and the local arrangements team is gratefully acknowledged.

At the general WIA meeting it was decided to rename WIA into *International Conference on the Implementation and Application of Automata* (CIAA) and to hold the first CIAA, that is, the fifth WIA, in London, Ontario, Canada, in the summer of 2000 in conjunction with the *Workshop on Descriptional Complexity of Automata, Grammars, and Related Structures* (DCAGRS) and a special day devoted to the 50th anniversary of automaton theory. The complete event would be called *Half a Century of Automaton Theory.*

March 2001 H. Jürgensen

Table of Contents

FA Minimisation Heuristics for a Class of Finite Languages

Jérôme Amilhastre, Philippe Janssen, and Marie-Catherine Vilarem

Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (UMR CNRS-Univ Montpellier II), IFA, 161 rue Ada 34392 Montpellier cedex 5, France
{amilhast,pja,mcvil}@lirmm.fr

Abstract. In this paper, we deal with minimization of finite automata associated with finite languages all the words have the same length. This problem arises in the context of Constraint Satisfaction Problems, widely used in AI. We first give some complexity results which are based on the strong relationship with covering problems of bipartite graphs. We then use these coverings as a basic tool for the definition of minimization heuristics, and describe some experimental results.

1 Motivations

Many AI problems can be expressed as Constraint Satisfaction Problems or CSP for short [Mon74]. A CSP involves a finite set of variables, a finite set of values for the variables and a set of constraints. Each constraint is defined as a relation on some subset of variables and gives the values which are mutually compatible for these variables. A solution is a value assignment to variables that satisfy all the constraints. Most of the CSP's works deal with the problem of computing one solution. Nevertheless, in some applications (e.g. design problems), it is necessary to compute and represent all the solutions.
In this paper we address the issue of representing and computing all solutions. One approach for this problem, proposed by Vempaty [Vem92], is to use Finite Automata (FA)[1]. Given a permutation of the variables set, the set of solutions appears as a regular language and can be represented by its Minimal Deterministic Finite Automata (MDFA). Solution sets languages (Ln) are finite sets words of equal length. Using FA allows incremental construction of the solutions set by applying classical operations on FA associated to each constraint. The efficiency of this method depends on the size of intermediate MDFA. Computation of MDFA recognizing finite languages has been studied in [Rev91,DWW98]. In this paper, we propose a more compact representation : Non Deterministic Finite Automata (NFA). It is well known that NFA may be exponentially more compact than equivalent DFA ; this property is preserved on the Ln class. However NFA minimization is an harder problem. The only studies on this problem concern the general case [Kim74,MP95,KW70] . In this paper we study this problem on the language class Ln.

[1] A similar approach using OBDDs has been used in order to represent boolean functions [Bry86].

2 Definitions

In the rest of this paper, we focus on FA recognizing Ln languages. We will use the following notations.

A finite state automaton $\mathcal{A}$ is a quintuple $(Q, \Sigma, \delta, I, F)$ where Q is a set of states, δ is the transition function $Q \times \Sigma \rightarrow 2^Q$, I and F are respectively the start and the accepting states. We consider only automata with an unique final state since such automata are sufficient to recognize Ln languages.
The *right language* (resp.*left language*) of a state q is $\mathcal{L}_{\mathcal{D}}(\mathcal{A}, q) = \{m \in \Sigma^* \;/\; F \in \delta^*(q, m)\}$ (resp. $\mathcal{L}_{\mathcal{G}}(\mathcal{A}, q) = \{m \in \Sigma^* \;/\; q \in \delta^*(I, m)\}$). In what follows, we will suppose that all states are accessible (their left language is not empty) and coaccessible (their right language is not empty). To denote the transition function, we will also use $\gamma^+_{\mathcal{A}}(q) = \{(d, q') \;/\; q' \in \delta(q, d)\}$ and $\gamma^-_{\mathcal{A}}(q) = \{(q', d) \;/\; q \in \delta(q', d)\}$.
Automata recognizing Ln languages have special properties. First, since Ln languages are finite, they are acyclic. Moreover, since all words are of equal length, set Q can be decomposed into levels. For a given state q, all the words of its left language have the same length i. We will say that i is the *level of* q and denote by $\mathcal{N}_{\mathcal{A}}(i)$ the set of states on level i. An automaton recognizing a Ln language $L \subseteq \Sigma^n$ has $n+1$ levels and is such that $\mathcal{N}_{\mathcal{A}}(0) = \{I\}$ and $\mathcal{N}_{\mathcal{A}}(n) = \{F\}$.

This work deals with the minimization of nondeterministic FA (NFA) and unambiguous FA (UFA). A UFA is an NFA in which there is a unique accepting computation for every accepted strings. The *Flower Automaton* is a distinguished automaton among all UFA recognizing the same Ln language.

Definition 1. *Let $L \subseteq \Sigma^n$, the* Flower Automaton *of L is a UFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ such that $\forall q \in Q \setminus \{I, F\}$ $\;|\gamma^+_{\mathcal{A}}(q)| = |\gamma^-_{\mathcal{A}}(q)| = 1$. It is the biggest UFA recognizing L.*

The use of NFA is motivated by the fact that NFA can be more compact than the equivalent MDFA. For general languages, the MDFA equivalent to a k-state NFA may have 2^k states. The following example shows that such a size difference also exists between UFA and MDFA recognizing Ln languages.
Let $\Sigma = \{1, 2, ..., n\}$ and $L_n = \bigcup_{j \in \Sigma} (\Sigma - j)^{n-1}.j \;\subset \Sigma^n$.
Consider the FA $\mathcal{A}_n$ with the following $n+1$ levels :
$\mathcal{N}_{\mathcal{A}}(0) = \{q_{0,1}\}$ $\mathcal{N}_{\mathcal{A}}(n) = \{q_{n,1}\}$ $\mathcal{N}_{\mathcal{A}}(i) = \{q_{i,1}, q_{i,2}, ..., q_{i,n}\}$ $(0 < i < n)$.
The $\mathcal{A}_n$ transition function is defined by :
$\forall i \in \{1, 2, ..., n-2\}, \forall p \in \{1, 2, ..., n\}, \forall j \in \Sigma - p,$
$\delta(q_{0,1}, p) = q_{1,p}$ $\;\delta(q_{i,p}, j) = q_{i+1,p}$ $\;\delta(q_{n-1,p}, p) = q_{n,1}$.
Figure 1 shows $\mathcal{A}_3$ and its equivalent MDFA. $\mathcal{A}_n$ is an UFA recognizing L_n. It has $n \times (n-1) + 2$ states. For the equivalent MDFA, the number of states is greater than 2^n since for each non empty proper subset of Σ there exists a state of $\mathcal{N}_{\mathcal{A}}(n-1)$.

3 Complexity of FA Minimization for *Ln* Languages

For general languages, FA minimization is PSPACE-hard [JR93]. In order to study the complexity of FA minimization for Ln languages, we introduce related problems on bipartite graphs.

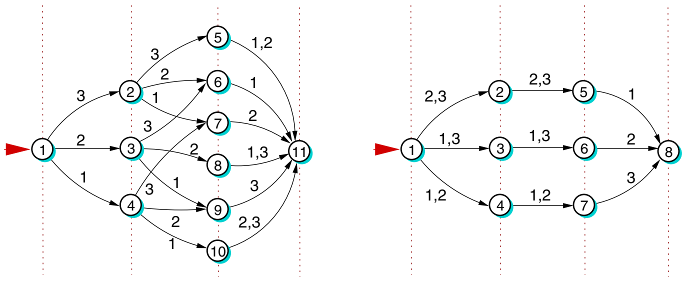


Fig. 1. $\mathcal{A}_3$ and the equivalent MDFA

A *bipartite graph* B is a finite, simple, undirected graph, given by (X_B, Y_B, E_B), where X_B et Y_B partition the vertices of B in two independent subsets, and E_B denotes the edges of B.
A *biclique* K of B is a pair (X, Y) such that $X \subseteq X_B$, $Y \subseteq Y_B$, X and Y are non-empty, and $B(X, Y)$, the subgraph induced by (X, Y), is complete ($X \times Y \subseteq E_B$).
A *biclique covering of* B is a set of bicliques $R = \{(X_1, Y_1), (X_2, Y_2), ..., (X_k, Y_k)\}$ such that $E_B = \bigcup_{(X_i,Y_i)\in R} X_i \times Y_i$.
A biclique cover is a *biclique decomposition of* B if and only if for every distinct bicliques K_i and K_j of R, K_i and K_j have no common edge.
We denote by RPB (resp. DPB) the decision problem associated with the computation of the minimum size of a biclique cover (resp. biclique decomposition) of B.

There is a strong relationship between biclique coverings of bipartite graphs and FA when all the words have length 2. We can associate a bipartite graph to each $L2$ language and there is a bijection between the FA (resp. UFA) recognizing this language and the bipartite biclique coverings (resp. decompositions). Figure 2 shows this relationship.

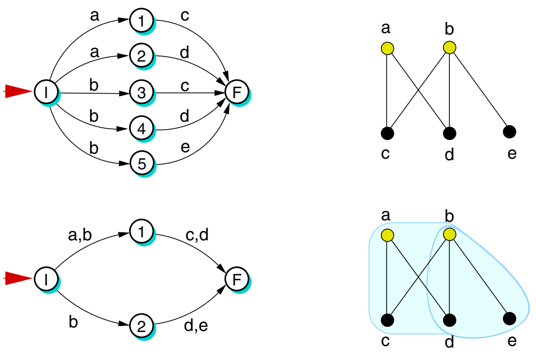


Fig. 2. Bipartite graph and corresponding Flower FA ; FA and corresponding biclique covering

We denote by (Flower FA $\rightarrow$ FA)-n (resp. (Flower FA $\rightarrow$ UFA)-n) the problem of constructing a minimum FA (resp. UFA) equivalent to a given Flower FA for *Ln* languages. In the following lemma, we also use (Flower FA $\rightarrow$ FA)-n and (Flower FA $\rightarrow$ UFA)-n for the associated decision problems.

Lemma 1. *(Flower FA $\rightarrow$ FA)-2 (resp. (Flower FA $\rightarrow$ UFA)-2) is polynomially equivalent to RPB (resp. DPB).*

Proposition 1. *The decision problems associated with (Flower FA $\rightarrow$ FA)-2 and (Flower FA $\rightarrow$ UFA)-2 are NP-complete.*

Proof. NP-completeness for (Flower FA $\rightarrow$ FA)-2 can be obtained by the previous lemma and the fact that RPB has been proved NP-complete [Orl77].
The DPB problem has been proved NP-complete by [AVJ98] ; it is an easy consequence of a result of [JR93] which states that obtaining a minimal UFA equivalent to a given DFA is NP-complete.

Corollary 1. *Minimization problems (FA $\rightarrow$ FA)-n, (FA $\rightarrow$ UFA)-n are NP-hard.*

4 FA Level Minimization Using Biclique Coverings

Because of the close relationship between biclique coverings of bipartite graphs and FA, we propose to use biclique coverings as a basic level minimization step. Let us first define the bipartite graph $\mathcal{B}_k$ that can be associated with each level k of a given FA $\mathcal{A}$ and the FA $\mathcal{A}/R_k$ that can be computed from $\mathcal{A}$ and any biclique covering of $\mathcal{B}_k$.

Definition 2. *Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a FA recognizing $L \subseteq \Sigma^n$ and k, $0 \leq k < n$ be a level of $\mathcal{A}$.*
• The bipartite graph of level k of $\mathcal{A}$ is $\mathcal{B}_k = (X_k, Y_k, E_k)$ with $X_k = \mathcal{N}_{\mathcal{A}}(k)$, $Y_k = \bigcup_{q \in \mathcal{N}_{\mathcal{A}}(k)} \gamma^+_{\mathcal{A}}(q)$ and $E_k = \{\{x, y\} \ / \ x \in \mathcal{N}_{\mathcal{A}}(k), y \in \gamma^+_{\mathcal{A}}(x)\}$.
• Let $R_k = \{(V_1, W_1), (V_2, W_2), ..., (V_p, W_P)\}$ be a biclique covering of $\mathcal{B}_k$. Then $\mathcal{A}/R_k$ denotes the FA obtained from $\mathcal{A}$ by replacing the states of $\mathcal{N}_{\mathcal{A}}(k)$ with $\mathcal{N}_{\mathcal{A}/R_k}(k) = \{V_i \ / \ (V_i, W_i) \in R_k\}$ and by updating the transition function to have $\gamma^+_{\mathcal{A}/R_k}(V_i) = W_i$ and $\gamma^-_{\mathcal{A}/R_k}(V_i) = \bigcup_{q \in V_i} \gamma^-_{\mathcal{A}}(q)$.

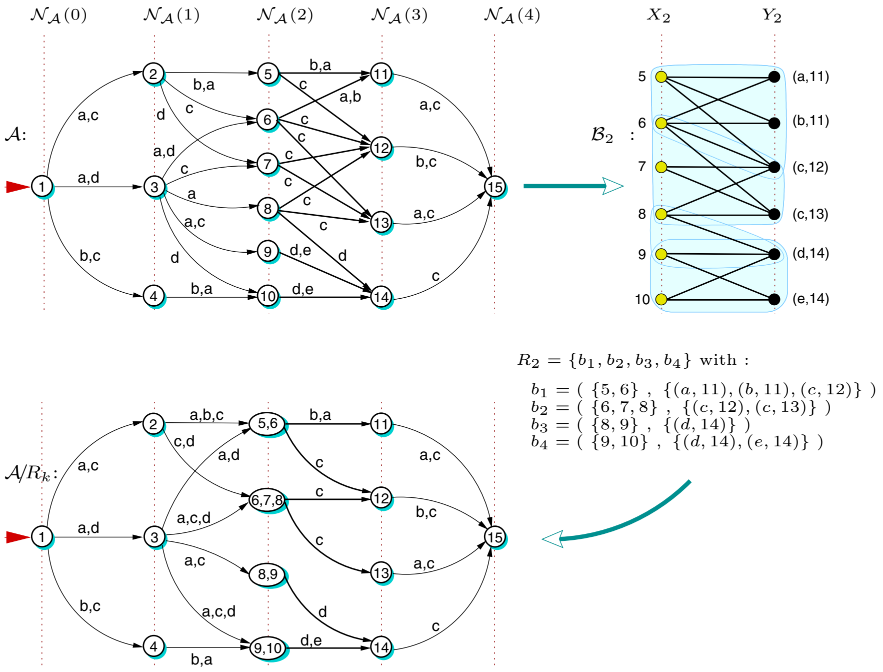
An example of FA $\mathcal{A}$ and $\mathcal{A}/R_k$ is given in Fig. 3 .

We can note that for an FA $\mathcal{A}$ and a biclique covering of $\mathcal{B}_k$, the FA $\mathcal{A}/R_k$ has $|R_k|$ states of level k while the other levels remain unchanged.

Let us introduce the following technical lemma from which some of the proofs in this paper can be directly obtained, as for example, the equivalence of $\mathcal{A}/R_k$ and $\mathcal{A}$.

Lemma 2. *Let $\mathcal{A}$ be a FA recognizing $L \subseteq \Sigma^n$, R_k be a biclique covering of $\mathcal{B}_k$ and let $\mathcal{A}'$ denotes the FA $\mathcal{A}/R_k$. We have then the following properties:*

(i) $\forall V \in \mathcal{N}_{\mathcal{A}'}(k)$, $\mathcal{L}_{\mathcal{G}}(\mathcal{A}', V) = \bigcup_{q \in V} \mathcal{L}_{\mathcal{G}}(\mathcal{A}, q)$
(ii) $\forall V \in \mathcal{N}_{\mathcal{A}'}(k)$, $\mathcal{L}_{\mathcal{D}}(\mathcal{A}', V) \subseteq \bigcap_{q \in V} \mathcal{L}_{\mathcal{D}}(\mathcal{A}, q)$
(iii) $\forall i \in \{0, 1, ..., n\}$, $i \not= k$, $\forall q \in \mathcal{N}_{\mathcal{A}'}(i)$, $\mathcal{L}_{\mathcal{G}}(\mathcal{A}', q) = \mathcal{L}_{\mathcal{G}}(\mathcal{A}, q)$.
(iv) $\forall i \in \{0, 1, ..., n\}$, $i \not= k$, $\forall q \in \mathcal{N}_{\mathcal{A}'}(i)$, $\mathcal{L}_{\mathcal{D}}(\mathcal{A}', q) = \mathcal{L}_{\mathcal{D}}(\mathcal{A}, q)$.


$\mathcal{N}_{\mathcal{A}}(0)$
$\mathcal{N}_{\mathcal{A}}(1)$
$\mathcal{N}_{\mathcal{A}}(2)$
$\mathcal{N}_{\mathcal{A}}(3)$
$\mathcal{N}_{\mathcal{A}}(4)$
X_2
Y_2
$\mathcal{A}$:
$\mathcal{B}_2$:
$\mathcal{A}/R_k$:
(a,11)
(b,11)
(c,12)
(c,13)
(d,14)
(e,14)

$R_2 = \{b_1, b_2, b_3, b_4\}$ with :

$b_1 = (\ \{5, 6\}\ ,\ \{(a, 11), (b, 11), (c, 12)\}\ )$
$b_2 = (\ \{6, 7, 8\}\ ,\ \{(c, 12), (c, 13)\}\ )$
$b_3 = (\ \{8, 9\}\ ,\ \{(d, 14)\}\ )$
$b_4 = (\ \{9, 10\}\ ,\ \{(d, 14), (e, 14)\}\ )$

Fig. 3. FA minimization from a biclique covering of the bipartite graph associated to level 2

Proof. (i) and (ii) can be directly deduced from definition of $\mathcal{A}'$

(iii) is verified for all levels smaller than k and we have (iii) for all levels greater than $k+1$ if and only if we have (iii) for level $k+1$.

(iv) is verified for all levels greater than k and we have (iv) for all levels smaller than $k-1$ if and only if we have (iv) for level $k-1$.

To show (iii) for the states of level $k+1$ and (iv) for the states of level $k-1$, we show that there is a path (q_{k-1}, q_k, q_{k+1}) such that $q_i \in \mathcal{N}_{\mathcal{A}}(i)$ labeled m in $\mathcal{A}$ if and only if there is a path from q_{k-1} to q_{k+1} labeled m in $\mathcal{A}'$.
$\Longrightarrow$ By definition of R_k, there is $(V, W) \in R_k$ such that $q_k \in V$ and $(m(k+1), q_{k+1}) \in W$. We then have by definition of $\mathcal{A}'$, $V \in \mathcal{N}_{\mathcal{A}'}(k)$ with $(q_{k-1}, m(k)) \in \gamma^-_{\mathcal{A}'}(V)$ and $(m(k+1), q_{k+1}) \in \gamma^+_{\mathcal{A}'}(V)$.
$\Longleftarrow$ Let (q_{k-1}, V, q_{k+1}) with $q_i \in \mathcal{N}_{\mathcal{A}'}(i)$ be a path of $\mathcal{A}'$ labeled m. By definition of $\mathcal{A}'$, there is some $q_k \in V$ such that $(q_{k-1}, m(k)) \in \gamma^-_{\mathcal{A}}(q_k)$ and, there is a pair $(V, W) \in R_k$ such that $(m(k+1), q_{k+1}) \in W$. As (V, W) is a biclique of $\mathcal{B}_k$, $(m(k+1), q_{k+1}) \in \gamma^+_{\mathcal{A}}(q_k)$.

Proposition 2. *Let $\mathcal{A}$ be a FA recognizing $L \subseteq \Sigma^n$ and R_k, a biclique covering of the bipartite graph $\mathcal{B}_k$ of $\mathcal{A}$. $\mathcal{A}/R_k$ recognizes L.*

Moreover, under some simple constraints on R_k, this transformation keeps the property of being deterministic or non ambiguous.

Proposition 3. *Let $\mathcal{A}$ be a DFA recognizing $L \subseteq \Sigma^n$. If R_k is a biclique covering of $\mathcal{B}_k$ such that $\forall (V_i, W_i), (V_j, W_j) \in R_k$ $(i \neq j)$ we have $V_i \cap V_j = \emptyset$, then $\mathcal{A}/R_k$ is a DFA.*

Proposition 4. *Let $\mathcal{A}$ be a UFA recognizing $L \subseteq \Sigma^n$. If R_k is a biclique decomposition of $\mathcal{B}_k$ then $\mathcal{A}/R_k$ is a UFA.*

Proof. Let $\mathcal{A}/R_k = \mathcal{A}' = (Q', \Sigma, \delta', I', F')$. If $\mathcal{A}'$ is ambiguous then there must be $V_1, V_2 \in Q'$ such that $\mathcal{L}_\mathcal{G}(\mathcal{A}', V_1) \cap \mathcal{L}_\mathcal{G}(\mathcal{A}', V_2) \neq \emptyset$ and $\mathcal{L}_\mathcal{D}(\mathcal{A}', V_1) \cap \mathcal{L}_\mathcal{D}(\mathcal{A}', V_2) \neq \emptyset$. From Lemma 2 we have $V_1, V_2 \in \mathcal{N}_{\mathcal{A}'}(k)$. Let $m \in (\mathcal{L}_\mathcal{G}(\mathcal{A}', V_1).\mathcal{L}_\mathcal{D}(\mathcal{A}', V_1)) \cap (\mathcal{L}_\mathcal{G}(\mathcal{A}', V_2).\mathcal{L}_\mathcal{D}(\mathcal{A}', V_2))$ and $q_k \in \mathcal{N}_\mathcal{A}(k)$ be the state of level k of $\mathcal{A}$ in the only path labeled m in $\mathcal{A}$. From (i) and (ii) of Lemma 2 we can deduce that $q_k \in V_1 \cap V_2$. Let $m = m_1 d m_2$ with m_1 of length k; by hypothesis on m, we have in $\mathcal{A}'$ two paths $I \xrightarrow{m_1} V_1 \xrightarrow{d} s_1 \xrightarrow{m_2} F$ and $I \xrightarrow{m_1} V_2 \xrightarrow{d} s_2 \xrightarrow{m_2} F$, where the states s_1 and s_2 belong to level $k+1$. As R_k is a biclique decomposition, $s_1 \neq s_2$. Moreover, by Lemma 2, the right and left languages of s_1 and s_2 are the same in $\mathcal{A}$ and $\mathcal{A}'$. We have thus constructed in $\mathcal{A}$ two distinct paths $I \xrightarrow{m_1 d} s_1 \xrightarrow{m_2} F$ and $I \xrightarrow{m_1 d} s_2 \xrightarrow{m_2} F$, which contradicts the hypothesis that $\mathcal{A}$ is an UFA.

5 FA Minimization Heuristics as a Sequence of Level Minimizations

We propose to minimize a FA by applying a finite sequence of our level minimization step. We first define efficient heuristics for the biclique covering problem and next the order in which the levels are minimized.

5.1 Biclique Covering Heuristics Inspired by Nerode Equivalence

There are many possible heuristics for the minimum biclique covering problem, which can in fact be formulated as a graph coloration problem [FH93]. The heuristic we propose is a generalization of the Nerode's well-known equivalence relation. Two states q and q' are *Nerode Equivalent* ($q \approx q'$) if and only if $\mathcal{L}_\mathcal{D}(\mathcal{A}, q) = \mathcal{L}_\mathcal{D}(\mathcal{A}, q')$. For our particular automaton, two states belonging to different levels cannot be Nerode Equivalent. Therefore Moore's classical recursive definition of this relation can be stated as follows : if the Nerode equivalence relation is equality for all pairs of states belonging to any level greater than k, then for all $q_1, q_2 \in \mathcal{N}_\mathcal{A}(k)$ we have $q_1 \approx q_2 \iff \gamma^+_\mathcal{A}(q_1) = \gamma^+_\mathcal{A}(q_2)$.

Let $\mathcal{A}$ be a FA recognizing $L \subseteq \Sigma^n$ and q_0, q_1 two different states of this FA such that $\gamma^+_\mathcal{A}(q_0) = \gamma^+_\mathcal{A}(q_1)$. We call *equality reduction* the operation consisting in modifying $\mathcal{A}$ in a new FA $\mathcal{A}'$ by removing the state q_0 and updating the transition function for q_1 in such a way that $\gamma^-_{\mathcal{A}'}(q_1) = \gamma^-_\mathcal{A}(q_1) \cup \gamma^-_\mathcal{A}(q_0)$.

It is then obvious that a MDFA can be computed from a given DFA $\mathcal{A}$ by applying successively all equality reduction operations on $\mathcal{A}$ from level $n-1$ down to level 1.

We next show that applying all the equality reduction operations on level k can be achieved by computing a particular biclique covering of the bipartite graph associated to this level.

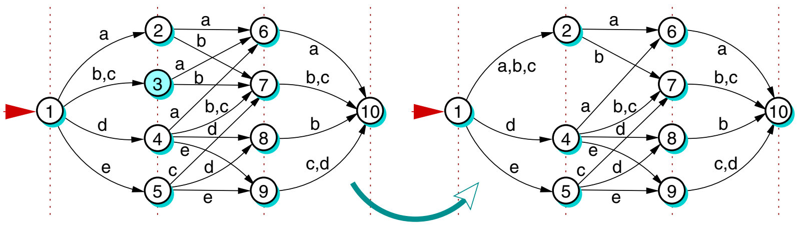


Fig. 4. Equality Reduction operation on state 3

Proposition 5. *Let $\mathcal{A}$ be a FA recognizing $L \subseteq \Sigma^n$. Let $\mathcal{T}$ be the function which computes from any given bipartite graph $B = (X, Y, E)$ the set of bicliques $\{(twins(x), N(x)) \ / \ x \in X\}$ where $N(x)$ denotes the neighborhood of x in B and where $twins(x)$ denotes the sets of vertices having the same neighborhood as x. Then for any level k of $\mathcal{A}$, $\mathcal{A}/\mathcal{T}(\mathcal{B}_k)$ is isomorphic to the FA obtained by applying all equality reduction operations on states of level k.*

We propose a generalization of the equality reduction operation and a function enabling to compute a biclique covering from a given set of such reductions.

Definition 3. *Given a FA $\mathcal{A}$, the couple $(q_0, \{q_1, ..., q_k\})$ is a* union reduction *if $q_0 \notin \{q_1, ..., q_k\}$ and $\gamma^+_{\mathcal{A}}(q_0) = \gamma^+_{\mathcal{A}}(q_1) \cup \gamma^+_{\mathcal{A}}(q_2) \cup ... \cup \gamma^+_{\mathcal{A}}(q_k)$. It is a* disjoint union reduction *if and only if $\gamma^+_{\mathcal{A}}(q_j) \cap \gamma^+_{\mathcal{A}}(q_i) = \emptyset$, $\forall q_j, q_i \in \{q_1, ..., q_k\}$. Applying this reduction to $\mathcal{A}$ consists in modifying $\mathcal{A}$ by removing state q_0 and updating the transition function for states $q_1, q_2, ..., q_k$ in such a way that $\gamma^-_{\mathcal{A}'}(q_i) = \gamma^-_{\mathcal{A}}(q_i) \cup \gamma^-_{\mathcal{A}}(q_0), \forall q_i \in \{q_1, q_2, ..., q_k\}$. k denotes the* degree *of the reduction.*

Figure 5 gives an example of a union disjoint reduction operation of degree 2 on the FA of the previous figure (a MDFA which was not reducible with the equality reduction operation).

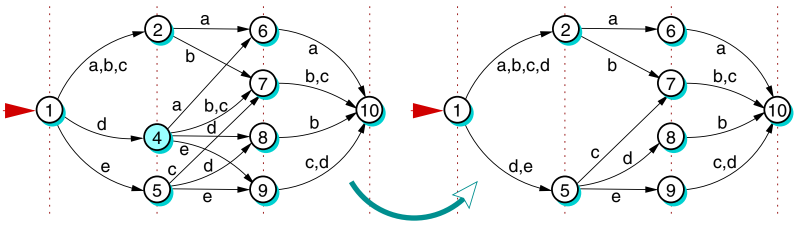


Fig. 5. Union Reduction operation on a MDFA

Note 1. For the sake of simplicity, we assume in the rest of this paper that there is no equality reduction operation on the considered FA levels.

To compute a biclique covering of $\mathcal{B}_k$ from a set of reductions, we use a function *Pred* defined by :

Definition 4. *Let $\mathcal{A}$ be a FA recognizing $L \subseteq \Sigma^n$ and $E = \{(q_1, S_1), (q_2, S_2), ..., (q_n, S_n)\}$ a set of union reduction operations on the states of level k of $\mathcal{A}$ such that the q_i are all distinct. Let $G(E)$ be the directed graph of vertices $\mathcal{N}_\mathcal{A}(k)$ whose edge set is $\{(q_i, q_j) \ / \ \exists \ (q_i, S_i) \in E, q_j \in S_i\}$. $\forall q \in \mathcal{N}_\mathcal{A}(k)$ let $Pred[q] = \{q\} \cup \{q' \in \mathcal{N}_\mathcal{A}(k) \ / \ \exists$ a directed path in $G(E)$ from q' to $q\}$. Then, $Rec(E) = \{(Pred[q], \gamma^+_\mathcal{A}(q)) \ / \ q \in (\mathcal{N}_\mathcal{A}(k) \setminus \{q_1, \cdots, q_n\})\}$.*

We then have the following Proposition:

Proposition 6. *Given a FA $\mathcal{A}$ recognizing $L \subseteq \Sigma^n$ and E a set of reductions on the states of $N \subseteq \mathcal{N}_\mathcal{A}(k)$, $Rec(E)$ is a biclique covering of $\mathcal{B}_k$ by $|\mathcal{N}_\mathcal{A}(k)| - |N|$ bicliques. It is a biclique decomposition if all the reductions in E are disjoint.*

Proof. • $Rec(E)$ is a biclique covering of $\mathcal{B}_k$.
Let $E = \{(q_1, S_1), (q_2, S_2), ..., (q_n, S_n)\}$ ($N = \{q_1, q_2, ..., q_n\}$). Let $C = (Pred[q], \gamma^+_\mathcal{A}(q))$ a pair of $Rec(E)$. If $Pred[q] = \{q\}$ then, C is clearly a biclique of $\mathcal{B}_k$. Otherwise, by definition of $Pred$, $\forall q' \in Pred[q]$, we have $\gamma^+_\mathcal{A}(q) \subseteq \gamma^+_\mathcal{A}(q')$ and C is also a biclique of $\mathcal{B}_k$. Let us show that each edge $e = (q, (d, q'))$ of E_k is covered by a biclique of $Rec(E)$. If $q \notin N$ the biclique $(Pred[q], \gamma^+_\mathcal{A}(q))$ in E covers e. Otherwise, let $P = \{p_1, p_2, ..., p_q\}$ be the set of sinks of $G(E)$ for which there is a path from q to p. It is then obvious from definition of $G(E)$ that $\gamma^+_\mathcal{A}(q) = \cup_{p \in P} \gamma^+_\mathcal{A}(p)$ and, as $R = \{(Pred[p], \gamma^+_\mathcal{A}(p)) \ / \ p \in P\} \subseteq Rec(E)$ and as $\forall p \in P$, $q \in Pred[p]$, there is a biclique of R covering e.
• If $\forall (q_i, S_i) \in E$, (q_i, S_i) is a disjoint reduction then $Rec(E)$ is a biclique decomposition of $\mathcal{B}_k$.
Let $K_0 = (Pred[q_0], \gamma^+_\mathcal{A}(q_0))$ and $K_1 = (Pred[q_1], \gamma^+_\mathcal{A}(q_1)) \in Rec(E)$. If K_0 and K_1 cover a same edge then $Pred[q_0] \cap Pred[q_1] \neq \emptyset$ and $\gamma^+_\mathcal{A}(q_0) \cap \gamma^+_\mathcal{A}(q_1) \neq \emptyset$. Let us show that this leads to a contradiction. Let us assume $Pred[q_0] \cap Pred[q_1] \neq \emptyset$ and let $q \in Pred[q_0] \cap Pred[q_1]$ be one of the closest states to q_0 and q_1. Let (q, S) be the reduction on q. Let $q'_0 \in Pred[q_0] \cap S$ and $q'_1 \in Pred[q_1] \cap S$. We have :

- $\gamma^+_\mathcal{A}(q_0) \subset \gamma^+_\mathcal{A}(q'_0)$ because $q'_0 \in Pred[q_0]$
- $\gamma^+_\mathcal{A}(q_1) \subset \gamma^+_\mathcal{A}(q'_1)$ because $q'_1 \in Pred[q_1]$
- $\gamma^+_\mathcal{A}(q'_0) \cap \gamma^+_\mathcal{A}(q'_1) = \emptyset$ because (q, S) is a disjoint reduction and $q'_0 \neq q'_1$.

We then have $\gamma^+_\mathcal{A}(q_0) \cap \gamma^+_\mathcal{A}(q_1) = \emptyset$.

Finally, level minimization consists in computing a maximum set of reduction operations E, computing the associated biclique covering $Rec(E)$ as in Definition 4 and then computing the reduced FA $\mathcal{A}/Rec(E)$. The next section provides a way to apply level reductions in order to compute a small FA.

5.2 Order of Level Minimization

For our languages, it follows from Proposition 5 that the minimization of a deterministic automaton can be done by computing a sequence of level minimizations. Moreover, Moore's recursive definition of the Nerode's equivalence leads directly to an efficient order for these level minimization steps : from right (greatest level) to left (smallest level). Keeping this order for our generalization leads to automata without reduction operations :

Assume that there is no reduction operation on any state of level greater than k. Assume that E is a maximum set of union reduction operations on states of level k of a FA $\mathcal{A}$. It then follows from the maximality of E and from Definition 4 that there is no union reduction operation on the states of level k of $\mathcal{A}/\mathcal{F}$.

On the one hand, it is clear that our algorithm can be very efficient. For example, it can be used to compute the small FA in Fig. 1 from the exponentially bigger equivalent MDFA. But, on the other hand it can fail to reduce some big automata. Consider the FA obtained from the MDFA in Fig. 1 by reversing its transitions (we will call it the reverse automaton). This FA clearly has no reduction operation and our minimization heuristic fails to reduce it when it can compute an exponentially smaller FA for the reverse automaton.

Now, let us call right reduction operations the reductions defined in 3 and define the left reduction operations. The couple $(q_0, \{q_1, ..., q_k\})$ is a left union reduction on a state q of $\mathcal{A}$ if $q_0, q_1, ..., q_k$ are different states of $\mathcal{A}$ such that $\gamma^-_{\mathcal{A}}(q_0) = \gamma^-_{\mathcal{A}}(q_1) \cup \gamma^-_{\mathcal{A}}(q_2) \cup ... \cup \gamma^-_{\mathcal{A}}(q_k)$. Let $\overleftarrow{\mathcal{A}}$ be the reversed automaton of the MDFA in Fig. 1 and let $\mathcal{A}'$ denote the reversed automaton of the smallest FA in this figure. $\overleftarrow{\mathcal{A}}$ is reducible with respect to left reduction operation and there is a sequence of such reductions which compute the exponentially smaller FA $\mathcal{A}$. But, because of the symmetry of the left and right reduction operation, $\mathcal{A}'$ can be obtained by reversing the FA computed by our right reductions based algorithm on $\mathcal{A}$.

As an efficient minimization algorithm with respect to left and right reductions should verify at least that the resulting FA has no left or right reduction operations, we propose to apply our heuristic to $\mathcal{A}$ and $\overleftarrow{\mathcal{A}}$. Nevertheless, a new right reduction operation can appear after a left reduction operation and vice-versa. Thus in order to compute a FA without right or left reduction operation, the final algorithm proposed in next section proceeds to more than one step of minimization of $\mathcal{A}$ and $\overleftarrow{\mathcal{A}}$, in fact as many as necessary.

5.3 Final Algorithm

Our FA minimization algorithm requires a function computing a maximum set of reduction operations. The choice of the reductions which it will be used to minimize a level is very important because it determines the reduction operations which will be possible to compute for the next level to minimize. From this point of view, it is a greedy algorithm : when choosing a set of reductions, we don't care what happens on other levels after applying it on a given level. There may be several union reductions for a given state, and for computational efficiency, we have to restrict the type of reductions (maximum degree fixed or not, disjoint or not). In fact, we have tried several possibilities and experimentally chose this one : computation of a *maximum set of disjoint reduction operations of degree 2*. This is in fact the smallest generalization of the equality reduction operations which can be used for DFA minimization. Note the pre-minimization of the level (line 1) removing equality reduction operations achieved before the computation of the reduction set (cf. Note 1).

Algorithm 1: MinimizeFA

```
    Data : A FA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ recognizing $L \subseteq \Sigma^n$
    $N \leftarrow |Q|$ ;
    repeat 2 times
        foreach level $k$ from $n-1$ to 1 do
1           $\mathcal{A} \leftarrow \mathcal{A}/\mathcal{T}(\mathcal{B}_k)$ ;
            $E \leftarrow$ maximum set of disjoint reductions of degree 2 ;
            $\mathcal{A} \leftarrow \mathcal{A}/Rec(E)$ ;
            $\mathcal{A} \leftarrow \overleftarrow{\mathcal{A}}$ ;
    if $|Q| \neq N$ then MinimizeFA($\mathcal{A}$)
```

Proposition 7. *Given a FA $\mathcal{A}$ recognizing $L \subseteq \Sigma^n$,* MinimizeFA($\mathcal{A}$) *is equivalent to $\mathcal{A}$, contains no left or right disjoint reduction of degree 2 or less and is a UFA if $\mathcal{A}$ is non ambiguous.*

6 Conclusion

We have run some experiments on a French dictionary and on pseudo-random languages. The French dictionary was changed in a *Ln* language by adding some special character * at the end of each word. The random *Ln* languages were generated with respect to the same fixed alphabet and language size with variable n. This results have been obtained on a Linux Pentium 350Mhz 64Mo RAM, times are given in seconds.

On one hand, these results are satisfying : UFA are 25% smaller in number of states than equivalent MDFA for the dictionary and, for random languages the gain in number of states goes to 50% down to 0 as the ratio number of words/language size decreases. But on the other hand, if we consider the only real parameter that reflects FA size i.e. the sum of the number of states and of the number of transitions, we must be much more reserved. Indeed, if the UFA can be smaller than the MDFA (up to 22% smaller), there are some languages for which it is larger, in the worst case (for random language of length 5) nearly 3 times larger. In fact, this can be explained by the choice of the biclique coverings : a reduction operation always decreases the number of states, but the number of transitions removed can be smaller than the number of transitions that appeared. It is clear that our general minimization scheme based on bipartite coverings is interesting, in particular because it enables us to compute non ambiguous FA. But it is also clear that the computation of the biclique coverings must take into account the number of transitions that will be created.

It seems that the number of states is always used to measure the size of automata. These experimentations shows that it would be better to also take into account the number of transitions by defining the size of an automaton as the sum of its number of states and number of transitions. This is clearly

Table 1. French dictionary ; $|Ln| = 58233, n = 26, |\Sigma| = 35$

	time	**#states**	**#st+#tr**
MDFA	7.54	29 819	88 487
UFA	45.71	23 346	84 992

Table 2. Random languages of variable length ; $|Ln| = 50000, |\Sigma| = 10$

	MDFA			UFA		
n	#states	#st+#tr	time	#states	#st+#tr	time
5	2133	18354	0.26	1121	51993	14
6	11022	62637	0.78	9355	68507	240
7	16423	78442	1.2	11069	72136	56
8	22204	94164	1.6	13552	77103	12
9	32017	114028	1.9	23853	97705	5.1
10	58585	167169	2.3	56773	163546	4.6
11	100147	250294	2.6	99839	249678	5.3
12	148307	346613	2.8	148273	346545	6.1
13	198073	446145	3.1	198069	446137	7
14	248076	546151	3.3	248075	546150	7.7

important from a practical point of view, but perhaps also from a theoretical one. We think that the following example illustrates this well.

Let $\Sigma_1 = \{1, 2, ..., 2^k - 1\}$, $\Sigma_2 = \{1, 2, ..., k\}$ and $f : \Sigma_1 \longrightarrow 2^{\Sigma_2} - \emptyset$ be a bijection between Σ_1 and $2^{\Sigma_2} \setminus \emptyset$. We defined then $L_k \subseteq \Sigma_1 \times \Sigma_2$ by $ij \in L_k \Longleftrightarrow j \in f(i)$. Let $\mathcal{A}$ be the FA of states $Q = \{I, F\} \cup \{q_1, q_2, ..., q_{2^k-1}\}$ and transitions : $\forall i \in \Sigma_1, \delta(I, i) = q_i$ and $\forall i \in \Sigma_2, \forall j \in f(i), \delta(q_i, j) = \{F\}$. $\mathcal{A}$ is the MDFA recognizing L_k. Let $\mathcal{A}'$ be the FA of states $Q' = \{I, F\} \cup \{q_1, q_2, ..., q_k\}$ and transitions : $\forall i \in D_1, \delta'(I, i) = \{q_j \ / \ j \in f(i)\}$ and $\forall i \in \Sigma_2, \delta'(q_i, i) = \{F\}$. $\mathcal{A}'$ is clearly a UFA recognizing L.

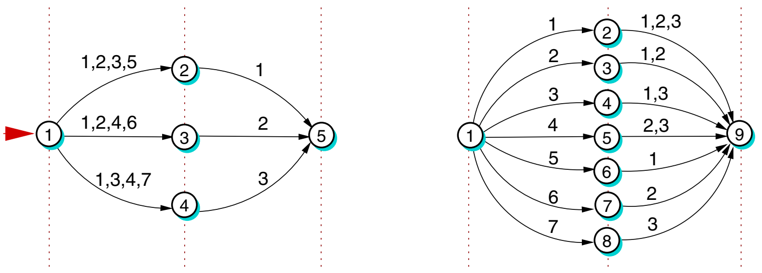


Fig. 6. An UFA and a MDFA recognizing L_3

It is then obvious that we have presented another example of a UFA exponentially smaller than the equivalent MDFA : $|Q'| = k + 2$ opposed to $|Q| = 2^k + 1$. But taking a look at number of states plus number of transitions (denoted by *RepSize*) leads to another conclusion : $RepSize(\mathcal{A}) = (2^k + 1) + (2^{k-1} + X)$ and $RepSize(\mathcal{A}') = (k+2) + (k+X)$ with $X = \frac{2^k \times k}{2}$ and then $RepSize(\mathcal{A}) \in \mathcal{O}(RepSize(\mathcal{A}'))$.

References

[AVJ98] J. Amilhastre, M. C. Vilarem, and P. Janssen. Complexity of minimum biclique cover and decomposition for bipartite domino-free graphs. *Discrete Applied Mathematics*, 86(2):125–144, September 1998.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C(35-6):677–691, 1986.

[DWW98] J. Daciuk, W. Watson, and B.W. Watson. Incremental construction of minimal acyclic finite state automata and transducers. In *Finite State Methods in Natural Language Processing*, Ankara, Turkey, 1998.

[FH93] P.C. Fishburn and P.L. Hammer. Bipartite dimensions and bipartite degrees of graphs. Technical Report 76, DIMACS, June 1993. available at http://dimacs.rutgers.edu/TechnicalReports.

[JR93] Tao Jiang and B. Ravikumar. Minimal NFA Problems are hard. *SIAM Journal of Computation*, 22:1117–1141, December 1993.

[Kim74] J. Kim. State minimization of nondeterministic machines. Technical report, IBM Thomas J. Watson Res. Center, 1974. RC 4896.

[KW70] T. Kameda and P. Weiner. On the state minimization of nondeterministic finite automata. *IEEE Trans. Comp.*, C(19):617–627, 1970.

[Mon74] U. Montanari. Networks of constraints : fundamental properties and applications to picture processing. *Information Sciences*, (7):95–132, 1974.

[MP95] O. Matz and A. Potthoff. Computing small nondeterministic finite automata. In A.Skou U.H. Engberg, K.G. Larsen, editor, *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 74–88. BRICS Notes Series, May 1995.

[Orl77] J. Orlin. Containment in graph theory: Covering graphs with cliques. *Nederl. Akad. Wetensch. Indag. Math.*, 39:211–218, 1977.

[Rev91] Dominique Revuz. *Dictionnaires et Lexiques, méthodes et algorithmes.* PhD thesis, Université Paris VII, Février 1991.

[Vem92] N. R. Vempaty. Solving constraint satisfaction problems using finite state automata. In *American Association for Artificial Intelligence (AAAI'92)*, pages 453–458, San Jose, 1992.

SEA: A Symbolic Environment for Automata Theory

Philippe Andary, Pascal Caron[⋆], Jean-Marc Champarnaud, Gérard Duchamp, Marianne Flouret, and Éric Laugerotte

LIFAR, Université de Rouen, 76134 Mont-Saint-Aignan Cedex, France

Abstract. We here present the system SEA which integrates manipulations over boolean and multiplicity automata. The system provides also self development facilities.

1 Introduction

The SEA symbolic environment arose from the dream of having within the same system different theories over automata facing each other, so that similarities and differences could be at hand. At the present stage, two theories offering minimization processes (however with some major differences, see below) were choosen: the boolean case and the case of multiplicities in a field (here $\mathbb{Q}$ to begin with). Many operations are common such as simple rational laws: union, concatenation, Kleene's closure (and their counterparts: sum, Cauchy product, star and external product) and extended rational laws: shuffle, Hadamard (intersection) and infiltration product.

A lot of softwares on automata have been developped. Let us cite automate [4], AMoRE [13] and Grail [16]. The package implementing boolean theory in SEA is the new version of the package AUTOMAP [3]. The advantages of the symbolic computation system Maple are numerous. The first one is in providing a simple and easy to use environment. The tight correspondance of the fundamental data types of Maple (lists, sets, tables) and the algebraic structure studied (automata, matrices) as well as the fact that the results of a computation can be used as input data for a subsequent computation have been good criteria of choice. The conversion of a mathematical expression into a Maple procedure is also quite natural; this feature implies a faster and more secure implementation than with a low-level language (such as C or C ++).

Many questions then arise and some major ones remain unsolved such as to find a deep link between the minimizations despite the fact that many features seem to be similar: minimal automaton of left quotients, isomorphism of minimal models, intertwining; the meaning of the star in the other boolean case ($\mathbb{K} = \mathbb{Z}/2\mathbb{Z}$); unicity of minimal models within certain classes of non deterministic automata. The structure of this short presentation is the following: in a first part (section 2), the theoretical background is presented as well as a description

[⋆] Corresponding author: `caron@dir.univ-rouen.fr` of the functions available in the environment, an example of computation is also provided, the second part is devoted to a technical description of the environment with a "how to" subsection, to end with we conclude by the description of some future steps as well as the setting of unsolved problems.

2 Theoretical Aspects of Boolean Automata and Automata with Multiplicities

2.1 Basic Definitions

Let A be a finite alphabet, and $\mathbb{K}$ a semiring. A formal series is a mapping S from A^* into $\mathbb{K}$ usually denoted by $S = \sum_{w \in A^*} \langle S|w\rangle w$ (where $\langle S|w\rangle := S(w) \in \mathbb{K}$ is the coefficient of w in S). The set of series, K^{A^*}, is naturally endowed with sum and external product. Concatenation is extended to series by the well defined formula $R.S = \sum_{w \in A^*} \left(\sum_{uv=w} \langle R|u\rangle\langle S|v\rangle \right) w$ and will be called convolution or Cauchy product in the case of multiplicities. Remark that, if $\langle S|1\rangle = 0$, then for every $w \in A^*$ the set $\{n/\langle S^n|w\rangle \neq 0\}$ is finite and, in this case $S^* := \sum_{w \in A^*} \sum_{n \geq 0} \langle S^n|w\rangle w$ is well defined. Notice that in case $\mathbb{K} = \{0,1\}$ (which occurs in two cases: $\mathbb{B}$ and $\mathbb{Z}/2\mathbb{Z}$), all functions are characteristic and series can be identified with subsets of A^*, the languages. In the boolean ($\mathbb{K} = \mathbb{B}$) case (resp. "with multiplicities") simple rational operations are union (resp. sum and external product), concatenation (resp. Cauchy product), Kleene's closure (resp. star). A language (resp. a series) is said rational iff it can be obtained from the letters by a (finite number of) combinations of the rational laws. The formula thus obtained is called a rational expression of S.

A (boolean) automaton over an alphabet A is usually defined [6,10] as a 4-tuple (Q, I, F, δ) where Q is a finite set of states, $I \subseteq Q$ the set of initial states, $F \subseteq Q$ the set of final states, and $\delta \subseteq Q \times A \times Q$ the set of edges. This feature can be extended to the case of multiplicities in any semiring. A $\mathbb{K}$-automaton [6] is then a 4-tuple (Q, I, F, δ) on a semiring $\mathbb{K}$, and the sets I, F and δ are rather viewed as mappings $I : Q \to \mathbb{K}$, $F : Q \to \mathbb{K}$, and $\delta : Q \times A \times Q \to \mathbb{K}$. In fact, a $\mathbb{K}$-automaton is an automaton with input weights, output weights, and a weight associated to each edge. Here, for each word $w = a_1 \cdots a_p \in A^*$, the coefficient $\langle S|w\rangle$ is the sum of the weights of paths labeled by $a_1 \cdots a_p$, this weight being obtained by the product of input, output and edges weights of the path. This is equivalent (with $n = |Q|$) to the data of a triplet (λ, μ, γ) where $\lambda \in \mathbb{K}^{1 \times n}$ codes the input states, $\gamma \in \mathbb{K}^{n \times 1}$ codes the output states, and $\mu : A \to \mathbb{K}^{n \times n}$ are the transition matrices, n being called the dimension of the representation. A series S is *recognizable* if it exists an automaton $\mathcal{A} = (\lambda, \mu, \gamma)$ such that $|\mathcal{A}| := \sum_{w \in A^*} \lambda\mu(w)\gamma w$ (the behavior of $\mathcal{A}$ [1]) is S. Schützenberger's classical theorem [17,8] asserts that rationnal series are exactly recognizable series. This is then an extension of Kleene's classical result [11]. The *left quotient* of a language L_2 with respect to a language L_1 is defined by $L_1^{-1}\cdot L_2 = \{v \mid u\cdot v \in L_2 \text{ and } u \in L_1\}$ or equivalently $\langle L_1^{-1}L_2|w\rangle := \langle L_2|L_1w\rangle$. The previous formula, well defined in the boolean case makes sense in general for L_1 with finite support. Thus, in any case, for a series (or a language) S, the left quotients $(w^{-1}S)_{w\in A^*}$ are defined and a variant of Schützenberger's theorem states: "S is rational iff it belongs to a finitely generated $\mathbb{K}$-module of series stable by the operators w^{-1}". In case the semiring is finite (in particular for $\mathbb{K} = \mathbb{B}$), this amounts to say that $\{w^{-1}S\}_{w\in A^*}$ is finite. In case $\mathbb{K}$ is a field, it is equivalent to the fact that the dimension of the $\mathbb{K}$-space generated by the family $(w^{-1}S)_{w\in A^*}$ is finite. Thus, in both cases, one defines the minimal automaton as the automaton with states $\{w^{-1}S\}_{w\in A^*}$ (resp. a basis of $Span((w^{-1}S)_{w\in A^*})$ in the case $\mathbb{K}$ is a field) and appropriate transitions. Finally, we should add that the minimal automaton computed in this way, is complete deterministic for the booleans, whereas it is minimal in the number of states among them recognizing the series, for the multiplicities in a field.

2.2 Operations over Automata

Simple rational operations. Operations over automata are deduced from that on series (i.e.: on languages for the boolean case). In both cases we have the classical rational operations which are *sum (union)*, *Cauchy product (concatenation)*, *star (Kleene's closure)* and external product. For details on the theory of series the reader is refered to [2].

Extended rational operations. There are other operations that preserve rationality. The implemented operations on automata corresponding to the shuffle ($\sqcup\!\sqcup$), Hadamard ($\odot$) [18] and infiltration ($\uparrow$) products [12] have been developped in [5], the behavior [6,1] of the resulting automata giving respectively the shuffle, Hadamard and infiltration products of the behaviors of the two automata. Let us remark that these three products can be defined from a double continuous family[1] of laws preserving rationality and satisfying the following recursion, for $a, b \in A$, $u, v \in A^*$,

$$\begin{cases} 1 \odot_{\epsilon,q} 1 = 1, \\ a \odot_{\epsilon,q} 1 = 1 \odot_{\epsilon,q} a = \epsilon a, \\ au \odot_{\epsilon,q} bv = \epsilon(a(u \odot_{\epsilon,q} bv) + b(au \odot_{\epsilon,q} v)) + q\delta_{a,b}a(u \odot_{\epsilon,q} v), \end{cases}$$

with $\delta_{a,b}$ the Kronecker symbol. With these notations, shuffle is $\odot_{1,0}$, Hadamard is $\odot_{0,1}$ and infiltration is $\odot_{1,1}$. The *intersection* of two $\mathbb{B}$-automata is then a specialization of Hadamard product.
Let us define mirror (unary operation, as the star operation), in order to state the proposition used for implementing the algorithm of the right quotient computation in the boolean case. Notice that the notation $S_1^{-1}\cdot S_2$ has not the same meaning as $L_1^{-1}\cdot L_2$ because it does not correspond in series to the same operations (quotient for languages, Cauchy product of inverse of S_1 and S_2 for series). The mirror of a $\mathbb{K}$-automaton is exactly the automaton in which the

[1] This family arises as a natural set of laws defined by duality [5].

initial states become final, the final states become initial and in which we reverse every edge (that is the automaton defined by the triplet $({}^t\gamma, {}^t\mu, {}^t\lambda)$). The *mirror* w^R of a word w over an alphabet A can be recursively defined as follows: $\epsilon^R = \epsilon$, and $(v\cdot a)^R = a\cdot v^R$ for $a \in A$ and $v \in A^*$. The mirror of $L \subset A^*$ is $L^R = \{w^R \mid w \in L\}$. The mirror of $S \in \mathbb{K}\langle\langle A\rangle\rangle$ is $S^R = \sum_{w\in A^*} \langle S|w\rangle w^R$. The *right quotient* of the language L_1 with respect to the language L_2, denoted by $L_1\cdot L_2^{-1}$, is dually defined by: $L_1\cdot L_2^{-1} = \{u \mid u\cdot v \in L_1 \text{ and } v \in L_2\}$, so one has

$$L_1\cdot L_2^{-1} = ((L_2^R)^{-1}\cdot L_1^R)^R.$$

2.3 Minimization

In both cases, automata (deterministic in the boolean case and any in case of a field) with the minimum number of states recognizing a given series are isomorphic to the minimal automaton previously defined [14] and it is well known that we have a minimization algorithm at our disposal (see [2,7] and [15,9] for details).

3 Presentation of the Packages

Several formats are used to represent automata. One for the multiplicity case and three for the boolean one. Functions to translate one automaton into another are provided. Two cases have to be distinguished (from multiplicity to boolean and conversely, and from boolean to boolean).

3.1 Description of the Formats

With the previous notations, (Q, I, F, δ) corresponding to the automaton of the figure 1 will be represented by either the *standard format*

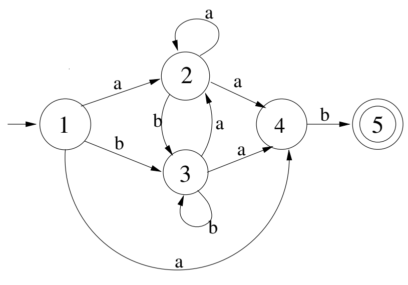


Fig. 1. $\mathbb{B}$-automaton

$$\begin{array}{lll}
[1,\{5\},\texttt{table}([& & \\
 & 1=\texttt{table}([&]), \\
 & a=\{2,4\} & 3=\texttt{table}([\\
 & b=\{3\} & a=\{2,4\} \\
 &]), & b=\{3\} \\
 & 2=\texttt{table}([&]), \\
 & a=\{2,4\} & 4=\texttt{table}([\\
 & b=\{3\} & b=\{5\} \\
 & &]) \\
 & &])]
\end{array}$$

– A single initial state (1)
– a set of final states ($\{5\}$)
– a transition table which is a table indexed on letters and on states ($table[1][a] = 2,4$ represents the two edges $(1,a,2)$ and $(1,a,4)$).

or the *list format*

$$\begin{array}{ll}
[1,\{5\},\texttt{table}([& \\
 & b=[\{3\},\{3\},\{3\},\{5\},\{\}], \\
 & a=[\{2,4\},\{2,4\},\{2,4\},\{\},\{\}] \\
])] &
\end{array}$$

which differs from the standard format only in the representation of the transition table. It is intended to provide an easy to use interface for users to enter their own automata. It is a table indexed by letters , each entry containing a list of sets which are a list of transitions for each states. The expression

```
t[a]=[{2,4},{2,4},{2,4},{},{}]
```

represents the edges $(1,a,2)$, $(1,a,4)$, $(2,a,2)$, $(2,a,4)$, $(3,a,2)$, $(3,a,4)$.

The *deterministic format* is the same as the list format (each sets of the transition table in the list format is replaced by the only state reached).

These three formats represents only boolean automata.

A $\mathbb{K}$-automaton has a matrix representation. The first element is the vector of entry cost in each state of the automaton, the last one the vector of exit costs, and the second one is the list of the transitions matrices corresponding to each letter. Hence, the list

$$\left[\begin{pmatrix} 2/3 & 0 & 0 & 0 \end{pmatrix}, \left[\begin{pmatrix} 0&1&0&0 \\ 0&0&1&0 \\ 0&0&0&0 \\ 0&0&0&0 \end{pmatrix}, \begin{pmatrix} 0&0&0&0 \\ 0&0&0&1 \\ 0&3&0&0 \\ 0&1&0&0 \end{pmatrix} \right], \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \right]$$

corresponds to the automaton given in figure 2.

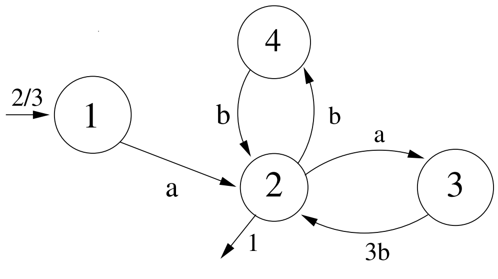


Fig. 2. $\mathbb{K}$-automaton

For the implementation of the AMULT package, we use the tables (as in the ABOOL package) which are dynamic structures more efficient than the matrix structures.

3.2 Operations on Automata

Both packages provide a set of operations on automata. These operations can be classified into three categories. The first one gathers operations on languages (series): concatenation (Cauchy product), union (sum), Kleene's closure (star),...). The table 1 gathers operations on automata which do not change the series that is recognized (determinization, minimization, reduction ...). The other operations are described in table 2. All functions corresponding to these operations have a shortcut (&C, &U, &S, ...). The last one is the set of manipulating operations. It gathers transformation format operations that are described in table 3 (deterministic format to standard one, standard format to lists format, multiplicity automaton to boolean one ...), as well as miscellaneous operations (states of an automaton , equivalence of two automata, ...).

Table 1. Transformation on automata

Boolean function	**Multiplicity function**	**Parameter**	**Description**
Deter	–	aut : automaton	returns the determinized automaton of aut
Mini	mini	aut : automaton	returns the minimal automaton of aut
Trim	–	aut : automaton	returns the reduced automaton of aut
Std	–	aut : automaton	returns an automaton with a single initial state without incoming edges
–	reduce	aut : automaton	returns an automaton in a right reduced form

Table 2. Operations on automata

Description	**Boolean function**	**Multiplicity function**	**Shortcuts**	**Parameters**
returns the complementary automaton w.r.t. the alphabet Σ	Comp	–		aut : automaton Σ : set of letters
returns the concatenation (Cauchy product) of the automata $aut1$ and $aut2$	Concat	Cauchy	&C, &c	$aut1, aut2$: automata
returns the automaton of aut^*	Star	star	&S, &s	aut : automaton
returns the automaton intersection (Hadamard product) of $aut1$ and $aut2$	Inter	Hadamard	&I, &i	$aut1, aut2$: automata
returns the automaton shuffle product of $aut1$ and $aut2$	Shuffle	shuffle	&W, &w	$aut1, aut2$: automata
returns the automaton mirror of aut	Mirror	–		aut : automate
returns the automaton difference of $aut1$ and $aut2$	Minus	–	&M	$aut1, aut2$: automata
returns the automaton of the n times concatenation	Power	–	&P	aut : automaton n : integer
returns the automaton of the left quotient of $aut2$ w.r.t. $aut1$	LeftQuotient	–	&L	$aut1, aut2$: automata
returns the automaton of the right quotient of $aut1$ w.r.t. $aut2$	RightQuotient	–	\$$R$	$aut1, aut2$: automata
returns the union(sum) automaton of $aut1$ and $aut2$	Union	sum	&U, &u	$aut1, aut2$: automata
returns the automaton external product of aut by α	–	ext		aut : automaton α : scalar
returns the automaton of the series S^{-1} s.t. $S\cdot S^{-1} = 1$	–	inv		aut: automaton of the series S
returns the automaton infiltration product of $aut1$ and $aut2$	–	ip		$aut1, aut2$: automata

Table 3. Miscellaneous functions

Tables of operations

Boolean function	**Multiplicity function**	**Parameter**	**Description**
Alphabet	–	*aut* : automaton	returns the alphabet of the automaton *aut*
Aut	–	*a* : letter	Build the automaton of the letter *a*
StToDf	–	*aut* : automaton	Builds a deterministic format transition table from a standard one
DfToSt	–	*aut* : automaton	Builds a standard format transition table from a deterministic one
StToLf	–	*aut* : automaton	Builds a lists format transition table from a standard one
LfToSt	–	*aut* : automaton	Builds a standard format transition table from a lists one
AreEquiv	–	$aut1, aut2$: automata	tests if the languages recognized by $aut1$ and $aut2$ are equal
States	–	*aut* : automaton	returns the set of states of *aut*
RecognizedBy	–	*w* : word *aut* : automaton	tests whether the word *w* is recognized by the automaton *aut*
BoolToMult	–	*aut* : automaton in a standard format	gives a matrix format for the AMULT package from a standard format
–	MultToBool	*aut* : automaton in a matrix format	give a standard format from a matrix format

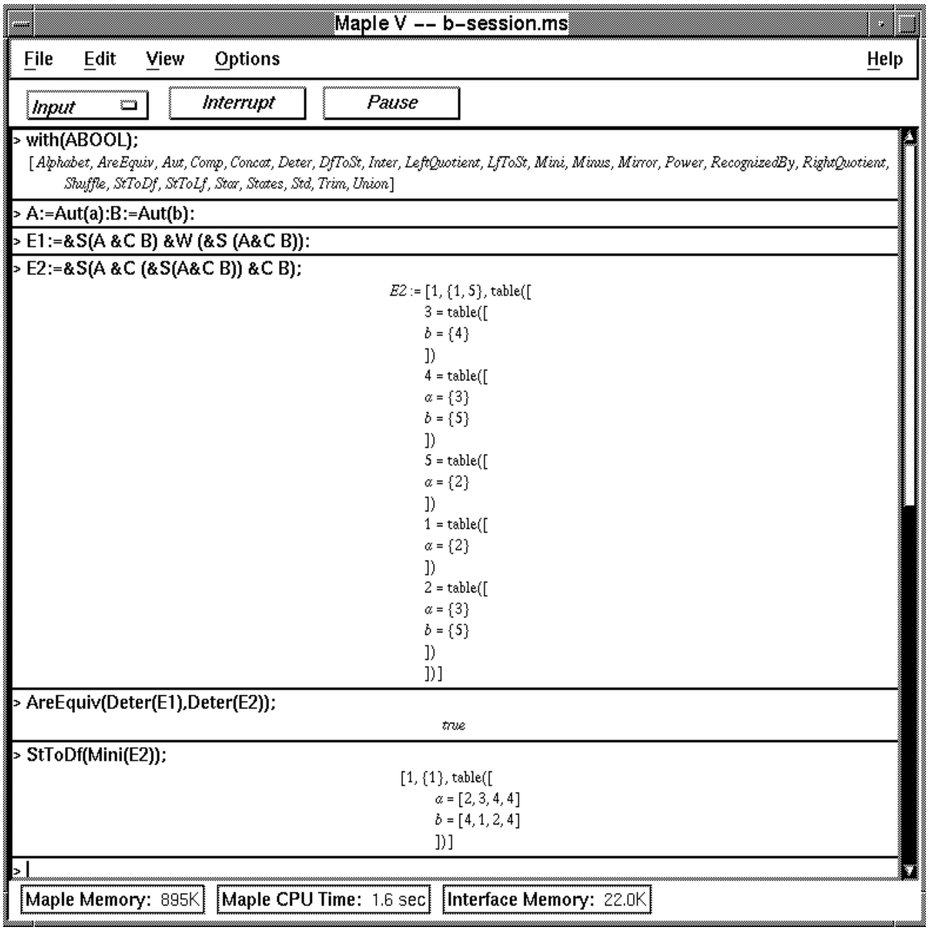
3.3 Example of Session

After loading the two packages ABOOL and AMULT, we present some possible computations. Consider first the two regular expressions $E_1 = (ab)^* \sqcup\!\sqcup (ab)^*$ and $E_2 = (a(ab)^*b)^*$, if the set of coefficients is the boolean semiring. Clearly, using the function `AreEquiv` of the package ABOOL, we find that they represent the same language. Indeed, we give the minimal automaton by the function `Mini`. However, if the set of coefficients is the field $\mathbb{Q}$, we can observe that multiplicities naturally appear in the linear representations issued of $r1$ and $r2$ associated respectively with E_1 and E_2. Using the fonction `mini` of the package AMULT, the resulted minimal linear representations are not isomorphic although the supports are equivalent (their regular expressions are respectively E_1 and E_2).

On this example, some last remarks can be made. In fact, the multiplicities being positive, we may substitute them by 1, and then we obtain an automaton in the boolean context. But, generally, this conversion is not immediate with the appearance of negative numbers. We must be carefull also that minimal linear representations with positive multiplicities does not imply the minimality (here in the number of states, the automaton may be not deterministic) of the corresponding automaton in the booleans. For example, if we take the $\mathbb{N}$-rational (or $\mathbb{Q}$-rational of course) series $S = \sum\limits_{w \in A*} F_{|w|+1} w$ where F_n is the n^th Fibonacci number, the minimal linear representation is

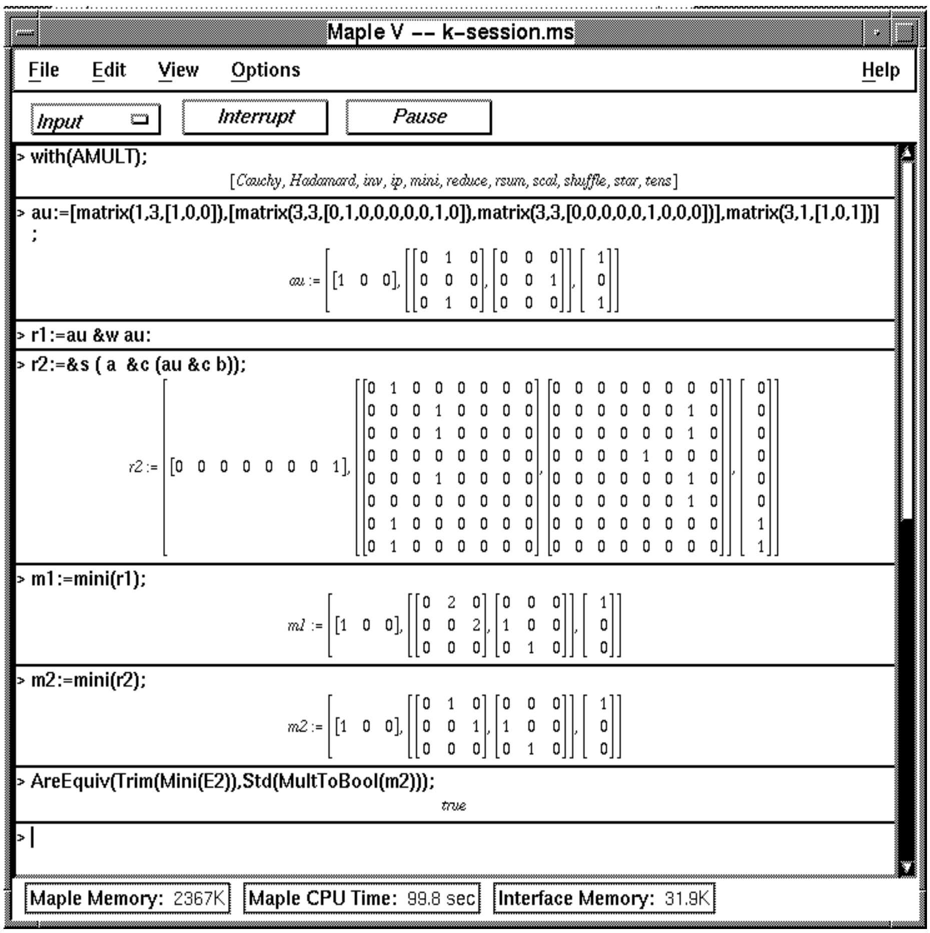
$$\left[(1\;0)\,,\left[\begin{pmatrix}0&1\\1&1\end{pmatrix},\ldots,\begin{pmatrix}0&1\\1&1\end{pmatrix}\right],\begin{pmatrix}1\\1\end{pmatrix}\right],$$

the transitions matrices being the same for each letter of the alphabet. But the minimal boolean automaton of the support (A^*) holds only one state.



```
> with(ABOOL);
  [Alphabet, AreEquiv, Aut, Comp, Concat, Deter, DfToSt, Inter, LeftQuotient, LfToSt, Mini, Minus, Mirror, Power, RecognizedBy, RightQuotient,
      Shuffle, StToDf, StToLf, Star, States, Std, Trim, Union]
> A:=Aut(a):B:=Aut(b):
> E1:=&S(A &C B) &W (&S (A&C B)):
> E2:=&S(A &C (&S(A&C B)) &C B);
                              E2 := [1, {1, 5}, table([
                                    3 = table([
                                    b = {4}
                                    ])
                                    4 = table([
                                    a = {3}
                                    b = {5}
                                    ])
                                    5 = table([
                                    a = {2}
                                    ])
                                    1 = table([
                                    a = {2}
                                    ])
                                    2 = table([
                                    a = {3}
                                    b = {5}
                                    ])
                                    ])]
> AreEquiv(Deter(E1),Deter(E2));
                                      true
> StToDf(Mini(E2));
                              [1, {1}, table([
                                    a = [2, 3, 4, 4]
                                    b = [4, 1, 2, 4]
                                    ])]
>
Maple Memory: 895K   Maple CPU Time: 1.6 sec   Interface Memory: 22.0K
```

The ABOOL package is first loaded. Indeed, we compute the automata of the two letters a and b, and then we build the automata of E_1 and E_2. The function AreEquiv allows us to verify that the two deterministic automata representing the languages $L(E_1)$ and $L(E_2)$, recognize the same language.



```
> with(AMULT);
                [Cauchy, Hadamard, inv, ip, mini, reduce, rsum, scal, shuffle, star, tens]

> au:=[matrix(1,3,[1,0,0]),[matrix(3,3,[0,1,0,0,0,0,0,1,0]),matrix(3,3,[0,0,0,0,0,1,0,0,0])],matrix(3,1,[1,0,1])]
  ;
                     [            [[0  1  0] [0  0  0]]  [ 1]]
              au := [[1  0  0], [[0  0  0],[0  0  1]], [ 0]]
                     [            [[0  1  0] [0  0  0]]  [ 1]]

> r1:=au &w au:

> r2:=&s ( a  &c (au &c b));
        [                             [[0 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 0]] [ 0]]
        [                             [[0 0 0 1 0 0 0 0] [0 0 0 0 0 0 1 0]] [ 0]]
        [                             [[0 0 0 1 0 0 0 0] [0 0 0 0 0 0 1 0]] [ 0]]
        [                             [[0 0 0 0 0 0 0 0] [0 0 0 0 1 0 0 0]] [ 0]]
  r2 := [[0  0  0  0  0  0  0  1],   [[0 0 0 1 0 0 0 0],[0 0 0 0 0 0 1 0]],[ 0]]
        [                             [[0 0 0 0 0 0 0 0] [0 0 0 0 0 0 1 0]] [ 0]]
        [                             [[0 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 0]] [ 1]]
        [                             [[0 1 0 0 0 0 0 0] [0 0 0 0 0 0 0 0]] [ 1]]

> m1:=mini(r1);
                     [            [[0  2  0] [0  0  0]]  [ 1]]
              m1 := [[1  0  0], [[0  0  2],[1  0  0]], [ 0]]
                     [            [[0  0  0] [0  1  0]]  [ 0]]

> m2:=mini(r2);
                     [            [[0  1  0] [0  0  0]]  [ 1]]
              m2 := [[1  0  0], [[0  0  1],[1  0  0]], [ 0]]
                     [            [[0  0  0] [0  1  0]]  [ 0]]

> AreEquiv(Trim(Mini(E2)),Std(MultToBool(m2)));
                                   true

> |

Maple Memory: 2367K   Maple CPU Time: 99.8 sec   Interface Memory: 31.9K
```

Next, the AMULT package is loaded. We compute the linear representations of r_1 and r_2. The implementation of the building of automata from the regular expression are identical for the two packages. Just it appears the external product in AMULT. After we obtain the minimal linear representations with the function mini, the conversion from the multiplicities into the boolean (multiplicities are replaced with 1) is made by the function MultToBool.

4 The SEA Environment

SEA is a tool provided for symbolic computation on automata under Maple system on UNIX platforms. The two major functionalities of SEA are automata Maple packages structuring and grouping, and `readlib` defined packages systematic generation.
When you develop a bunch of Maple functions you finally group and structure

them into packages, because doing this way you can write a more easily maintained, reusable code.

Maple enables the developer to create simple packages which are just tables in which the indices are function names and the entries function bodies. Saving this table into a "`.m`" file in the correct location permits anyone to load the package further during another Maple session, using the `with` command.

When the functions are cumbersome and numerous, you may want to load them only when needed, rather than systematically at the beginning of the session. This can be achieved via the `readlib` Maple command in the following way. For every index named `f` in the table package, remove its entry and replace it with the following protected form:

```
'readlib('f', <file>)':
```

where `<file>` is the absolute name of the file containing a definition for your `f` function (and possibly other things). Thus loading the table package is done very quickly because it does nothing but defining where your functions are located. At first call, your function will be loaded and the remember table of the `readlib` command prevents the system to load it again.

4.1 How to Write Your Package

First of all, you have to write down your functions. We distinguish between two kinds of functions: those that make up the public part (the interface) of the package, accessible for the client user, and those that are somehow private and thus theoretically inaccessible[2] for the client user. We call *external* the first kind of functions, and *internal* the second one. Notice that you can refine again the second category of functions by defining those that are used by only one external function (*satellite* functions), and those that are used by many (*tool* functions).

For the package to be successfully managed by SEA, you have to ensure that your source code satisfy the following requirements.

1. There is only one external function (and possibly many satellite functions) per source file (generally named according to the external function).
   – The signature of the external function begins with the following regular expression within its source file
     ```
     ^[\t, ]*`<pnam>_EXT/<fnam>`[\t, ]*:=[\t, ]*proc[\t, ]*(.*$
     ```
     where `<pnam>` and `<fnam>` are the package name and the external function name respectively. Notice that the parameters list can extend over many lines.
   – The signature of internal functions begins with the following regular expression within the source file
     ```
     ^[\t, ]*`<pnam>/<fnam>`[\t, ]*:=[\t, ]*proc[\t, ]*(.*$
     ```

[2] In fact there is no information hiding in Maple so that you can ignore the privacy of these functions; but they may be undocumented, meaning that you do not have to use them.

  - Every function call must be in Maple's long format:
    - use `<pnam>[<fnam>](<parlist>)` to call the external function `<fnam>` with the `<parlist>` list of parameters, lying in the `<pnam>` package,
    - use `` `<pnam>/<fnam>`(<parlist>) `` to call the internal function `<fnam>` with the `<parlist>` list of parameters, lying in the `<pnam>` package.

2. There is one file per help function (which looks like `` `help/text/<fnam>` := TEXT(...): ``), and one help function for every external function and for the package itself.
3. There is only one file (named `declext`) for all the external constants and variables, which will appear in the package table.
4. There is only one file (named `declint`) for all the local types and tool functions, which will not appear in the package table.

Maple documentation tells us that we can write a `` `<pname>/init` `` function which is automatically executed before any `with` command call, until the (global) variable `` `<pname>/Initialized` `` becomes `true`. You can use this trick to do some initial work prior to execute any of your package function.

Every file you provide will be precompiled "as is" into Maple internal format so that you can put comments into your source or help files safely.

4.2 How to Use SEAER Package

You are given an error package to manage errors from one place, providing a standard output format for all error messages.

The package is named SEAER and it contains (at the present stage) a unique function named `SEAER[SeaErr]`.
Its standard calling sequence is `SeaErr(name, errno)` where `name` is the name of the client function (*i.e.* its `procname`) and `errno` is the identifier of the detected error. At the moment, there is only two public error identifiers

- `SEAER[INP]` meaning *Invalid Number of Parameters*, and
- `SEAER[IPT]` meaning *Invalid Parameter Type.*

Further error identifiers may be defined on request to the technical administrator[3].

[3] We stress the fact that you can use SEA in two different manners. On the one hand, you can use SEA freely for your own sake: compile any kind of Maple package (even not related to automata theory), use your own error function(s) or adapt the SEAER package to your convenience, modify any sources, and so forth. On the other hand, if one wants to add functionalities within SEA, he (or she) has to make a proposal to the administrator. Anyhow, distribution under the name SEA is only allowed when the original unmodified SEA package is concerned.

```
$SEAPATH
  +- CONTRIB
  |    +- ABOOL
  |    +- AMULT
  |    +- YRPKG
  |    |    +- DOC
  |    |    |    +- YRPKG-usr.ps
  |    |    |    +- YRPKG-ref.ps
  |    |    |    +- readme.1st
  |    |    +- SRC
  |    |    |    +- ENVT
  |    |    |    |    +- declext
  |    |    |    |    +- declint
  |    |    |    +- FCTN
  |    |    |    |    +- FirstFun
  |    |    |    |    +- LastFun
  |    |    |    +- HELP
  |    |    |         +- FirstFun
  |    |    |         +- LastFun
  |    |    |         +- YRPKG
  |    |    +- XMPL
  |    |         +- FirstXmpl.m
  |    |         +- SecndXmpl.m
  |    +- SEAER
  +- NEWS
  +- SEASTEM
```

4.3 How to Incorporate Your Package into SEA

When you are ready with your source code, you can put your package (say `YRPKG`) into SEA.

First you have to set the `SEAPATH` environment variable to the directory where SEA resides (for example `/home/lname/Maple/SEA`). Then you create a tree structure in the `$SEAPATH/CONTRIB` directory containing all your source files (Maple code and documentation). Now you have to install your package, that is you have to precompile it into Maple internal format, build the package files, and place them into your local library. This is quite simple because you just have to command

```
$SEAPATH/SEASTEM/UTL/inst [-v VerboseDir] MapleLibDir [PkgName]*
```

The `v` option means verbose, so you can look what `inst` really does for you in the `VerboseDir` directory. The `MapleLibDir` parameter is the target directory for all your compiled files (for example `/home/lname/Maple/Lib/SEA`). Finally you give the name(s) of the package(s) you want to (re)install, none means all.

Here you are! Now you can run Maple and load any package you want, or simply command `with(SEA):` at the prompt.

5 Conclusion

The joint development of the two packages yielded a two way interaction between theory and implementation. This work has generated many theoretical questions closely related to the two theory in presence.

Moreover, we have seen that extended rational laws are better understood (and implemented) as dual laws deriving from suitable co-products. The compatibility of congruences with these coproducts have been completely solved in certain cases, the general problem seems to be manifold and remains difficult. In particular, among next steps we plan to implement finite semirings with some new features coming from p-combinatorics.

References

1. J. Berstel and D.Perrin, *Theory of codes*, Academic Press, 1985.
2. J. Berstel and C. Reutenauer, *Rational series and their languages*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, 1988.
3. P. Caron, *AG: A set of Maple packages for manipulating automata and finite semigroups.* Software–Practice & Experience, **27(8)**, 863–884, 1997.
4. J.-M. Champarnaud and G. Hansel, *Automate, a computing package for automata and finite semigroups*, J. Symbolic Comput., **12**, 197–220, 1991.
5. G. Duchamp, M. Flouret and E. Laugerotte, Operations over automata with multiplicities, WIA 98, to appear in Lect. Notes in Comp. Sci.
6. S. Eilenberg, *Automata, Languages, and Machines*, Vol. A, Academic Press, 1974.
7. M. Flouret and E. Laugerotte, Noncommutative minimization algorithms, Inform. Process. Lett., **64**, 123-126, 1997.
8. M. Flouret, Contribution à l'algorithmique non commutative, Ph.D. thesis, University of Rouen, 1999.
9. J. E. Hopcroft, *An n log n algorithm for minimizing the states in a finite automaton*, In Z. Kohavi, editor, The theory of machines and computations, pages 189–196. Academic Press, New York, 1971.
10. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.
11. S. C. Kleene, Representation of events in nerve nets and finite automata, Automata studies, 1956.
12. M. Lothaire, *Combinatorics on words*, Addison-Wesley, 1983.
13. O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkena, *Report on the program AMoRE*, Report, Institut für informatik und praktische mathematik, Christian-Albrechts Universität, Kiel, 1995.
14. E. F. Moore, *Gedanken experiments on sequential machines*, In Automata studies, pages 129–153, Princeton Univ. Press, Princeton, N.J., 1956.
15. A. Nerode, *Linear automata transformation* In Proceedings of AMS 9, pages 541–544, 1958.
16. D. Raymond and D. Wood, *Grail, a C++ library for automata and expressions*, J. Symbolic Comput., **17**, 341–350, 1994.
17. M. P. Schützenberger, On the definition of a family of automata, Inform. and Control, **4**,245-270, 1961.
18. M. P. Schützenberger, On a theorem of R. Jungen, Proc. Amer. Soc. , **13**, 885-890, 1962.

Analysis of Reactive Systems with n Timers

Anne Bergeron and Riccardo Catalano

LACIM, Université du Québec à Montréal,
C.P. 8888 Succursale Centre-Ville, Montréal, Québec,
Canada, H3C 3P8,
anne@lacim.uqam.ca

Abstract. In this paper, we develop theoretical, as well as practical, tools for the synthesis and the verification of processes that contain n timers. Such tools are equally adapted to numerical calculations as to symbolical ones, thus allowing for parametric analysis.
The results we have obtained rely on a simple and efficient representation of the states of an automaton that recognizes the behaviors of the process. This representation is based on a *mechanical structure* which helps us encode the states in a compact manner and leads to simple algorithms.

Keywords: automata, real-time systems, verification, synthesis.

1 Introduction

We are interested in reactive systems whose behavior depends on both the events of a process and the *start* and *timeout* events of n timers. The timers are started individually or in groups with the occurence of an event. The timeout event of a given timer can, in turn, influence the behavior of the process. This kind of system can be found in all applications where a given delay between two actions or events must be met or exceeded. Transportation or telecommunication networks are good examples of such systems.

Consider, for example, a simple process such as the one modeled in Fig. 1.

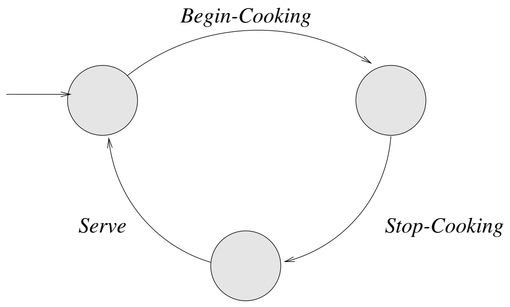


Fig. 1. *A simple process*

This automaton describes the *logical* sequences of events of the process, but does not capture any timing constraint between them. These must be stated as additional properties such as:

(1) The events *Begin-Cooking* and *Stop-Cooking* must be separated by a delay of at least d_1 minutes.

(2) The events *Begin-Cooking* and *Serve* must be separated by a delay of at most d_2 minutes.

One way to specify this type of constraint is to associate with each constraint a *symbolic timer.*

Definition 1. *A symbolic timer* **T** *is given by a real positive constant* d*, and by the following automaton:*

$$\begin{tikzcd}
\textit{inactive} \arrow[loop above, "-"] \arrow[rr, bend left, "S"] & & \textit{active} \arrow[loop below, "-"] \arrow[ll, bend left, "T"]
\end{tikzcd}$$

■

In its initial state, the timer is *inactive.* Event S starts the timer. After a delay d, event T occurs and the timer returns to its initial state. In each state, null loops labeled '–' allow for the synchronization with external events. Thus a timer models the language of alternating events S and T, possibly interspersed with other events from external processes.

When two timers function in parallel, the possible events can be described by vectors of the form:

$$\begin{bmatrix} e_1 \\ e_2 \end{bmatrix}$$

where e_1 is an event of the first timer, and e_2 is an event of the second timer. Using this notation, the sequence of timer events that would be associated with a correct sequence of events of Fig. 1 would be:

$$\begin{array}{cccccc}
 & \begin{array}{c}\textit{Begin-}\\ \textit{Cooking}\end{array} & & \begin{array}{c}\textit{Stop-}\\ \textit{Cooking}\end{array} & \textit{Serve} & \\
\begin{array}{c}\mathbf{T}_1\\ \mathbf{T}_2\end{array} & \begin{bmatrix} S \\ S \end{bmatrix} & \begin{bmatrix} T \\ - \end{bmatrix} & \begin{bmatrix} - \\ - \end{bmatrix} & \begin{bmatrix} - \\ - \end{bmatrix} & \begin{bmatrix} - \\ T \end{bmatrix}
\end{array}$$

With this simple example, we see that such a sequence is possible if and only if $d_1 < d_2$ where d_1 is the delay of timer 1 and d_2 is the delay of timer 2.

Note that this type of condition on the delays depends only on the sequence of timer events, and not on the events of the original process. We will thus set aside, for the time being, all external processes and focus our attention on the possible event sequences of a system of n timers. Let us assume that the delays of the n timers are given by

$$D = (d_1, \ldots, d_n).$$

We will consider the following problems:

(1) *Verification*: Given the delays $D = (d_1, \ldots, d_n)$, what are the *possible* event sequences? Is the set of all such sequences a regular language?

(2) *Synthesis*: What constraints must be placed on the vector D in order for a given set of behaviors to be possible? Or impossible?

In order to study these problems, several models have been put forth, the best known of which is undoubtedly that of *timed automata* [1]. This theory notably shows that behaviors are recognizable by finite automata in the case of rational delays. Moreover, in [4], we find a proof of recognizability in the case of $n = 2$, for all real delays. In the following paragraphs, we develop techniques that allow for the verification of systems with parametric delays.

Problem number (2) is somewhat more difficult [2], [3]. A systematic process must be developped which will generate timing constraints ensuring that a given set of behaviors is possible or impossible. We will introduce algorithms that provide simple decision making procedures for the enumeration of constraints.

2 Event Sequences on n Timers

A system of n timers with delays $D = (d_1, d_2, \ldots, d_n)$ is composed of timers that function in parallel.

An event of the global system of timers is a vector whose components belong to the set $\{S, T, -\}$. We are interested in the possible behaviors of systems of n timers and therefore in the set of event sequences. For example, the sequence *abcde*:

$$\begin{array}{c|c|c|c|c|c} & a & b & c & d & e \\ \mathbf{T}_1 & S & - & - & - & T \\ \mathbf{T}_2 & S & - & T & S & - \\ \mathbf{T}_3 & - & - & S & - & - \\ \mathbf{T}_4 & - & S & - & - & - \\ \mathbf{T}_5 & - & - & - & - & S \end{array}$$

is defined on a system of 5 timers.

Such a sequence is said to be *possible* if we can place its events on a *time line*, like the one in Fig. 2, such that the following conditions hold:

(1) The events are separate.
(2) The distance between the event in which timer i is started and the event in which it times out equals d_i.
(3) If timer i is still running after the sequence of events, the distance between the event that last started it and the *reading* point is strictly less than d_i.

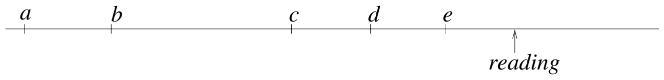


Fig. 2. *A time line associated with the sequence abcde.*

For a given time line, the *readings* of the timers are given by the vector

$$R = (r_1, r_2, \ldots, r_n)$$

where r_t is the distance between the event that last started the timer and the reading point. The value of r_t for inactive timers is irrelevant and is set to 0.

If we know the value of R, we can determine the possible next events of the system: in fact, we can predict which timer will time out next.

In general, there exists an infinite number of time lines associated with a given sequence y. Thus, if the set of readings that corresponds to the set of time lines is known, we can predict all possible next events of the system.

Definition 2. *Let y be a sequence of events on n timers, the* state S_y *is the set of all possible readings after the sequence y.*

It is well known [5], [1] that a state S_y is a convex polytope of dimension $k \leq n'$, where n' is the number of timers that are still running after the sequence y. Representations of such states are often complex and difficult to manage. In the following sections, we will assign a mechanical representation to a state, allowing us to develop simple algorithms for computing states and transitions.

3 A Mechanical Interpretation of a State

The distances between events of a sequence are bound by certain constraints. For example the distance between an event that starts a timer and the event in which it times out is constant for all possible time lines. In general, we have:

Definition 3. *Let $e_1e_2 \ldots e_m$ be a possible event sequence. Two events of the sequence are said to be* linked *if the distance that separates them is the same in every time line.*

It is the equivalence relation generated by the pairs of events

$$(e_k, e_l)$$

where e_k is an event that starts timer i and e_l the next event in which timer i times out. In the example given earlier:

$$\begin{array}{cccccc} & a & b & c & d & e \\ \begin{matrix}\mathbf{T}_1\\ \mathbf{T}_2\\ \mathbf{T}_3\\ \mathbf{T}_4\\ \mathbf{T}_5\end{matrix} & \begin{bmatrix}S\\ S\\ -\\ -\\ -\end{bmatrix} & \begin{bmatrix}-\\ -\\ -\\ S\\ -\end{bmatrix} & \begin{bmatrix}-\\ T\\ S\\ -\\ -\end{bmatrix} & \begin{bmatrix}-\\ S\\ -\\ -\\ -\end{bmatrix} & \begin{bmatrix}T\\ -\\ -\\ -\\ S\end{bmatrix}\end{array}$$

the events $\{a, c, e\}$ form an equivalence class because of the pairs (a, e) in which timer 1 is started and times out, and the pair (a, c) in which timer 2 is started and times out.

The above definition allows us to introduce a representation of a time line that captures the essential constraints on the positions of the events of a sequence (Fig. 3).

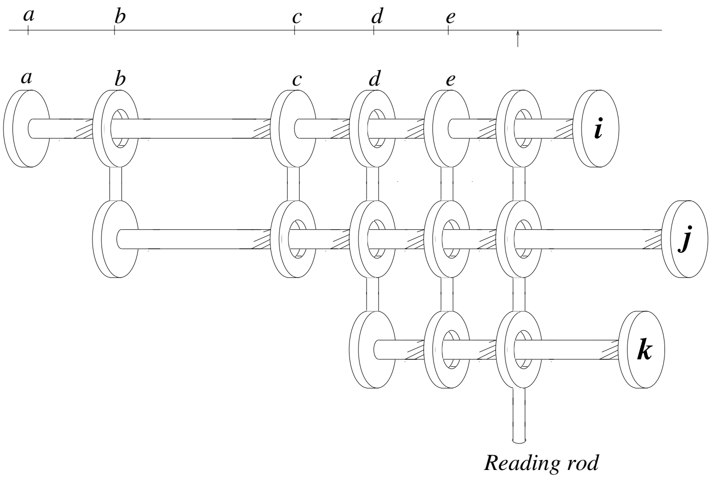


Fig. 3. *A state.*

In this structure, each event of a sequence defines a vertical rod possessing eyelets, and welded to one horizontal rod. Linked events are welded to the same horizontal rod thus, a horizontal rod represents an equivalence class of events. The right end of a horizontal rod represents the *next event* of the rod, i.e. the moment when the next timer belonging to the rod will time out.

Apart from its illustrative qualities, the representation of Fig. 3 allows us to reason in an analog way on the properties of a state. For example, it isn't hard to convince ourselves of the following:

Observation 1 *There is always a certain play between two horizontal rods.*

Since the events are separated, two rods will be separated by the minimal distance between pairs of events of the two corresponding classes.

Observation 2 *By sliding the rods horizontally, we can obtain all possible readings for a given state.*

Each possible time line coressponds to a position of the mechanical structure. And one time line can be obtained from any other by modifyng the distance between events that are not in the same class, that is, sliding a rod with respect to another.

Observation 3 *There exists one and only one position of the structure in which each timer has a minimal reading: it is obtained by pushing all the rods to the right, against the vertical rod representing the reading point. This position is called the* minimal position*.*

Another way to obtain this minimal position is to rotate the structure clockwise for 90^o while holding the reading rod, and let gravity do its work.

The structure captures all the essential caracteristics of a state. For the time being, such a construct depends on the complete knowledge of an event sequence. It is therefore of little use for the construction of an automaton recognizing the set of possible event sequences. However, in the following section, we will see how all the possible movements of this structure can be encoded by a small set of values. Given a state and a possible event, the next state of the automaton will result from simple *mechanical* operations on the rods.

4 Description of a State

A state must contain sufficient information to predict which events are possible, and what should be the next state after any possible event. In this section, we will show that the following numbers do indeed describe a state.

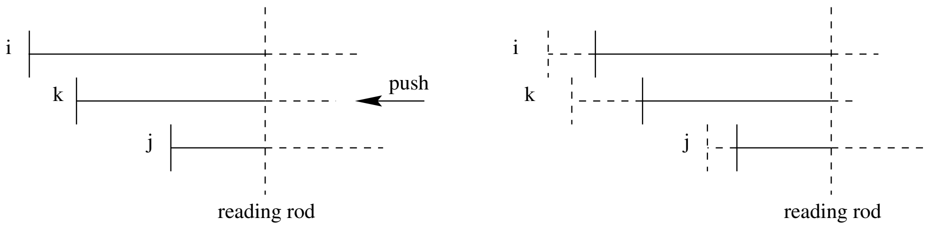
Let S_y be a state, that is a set of possible readings for n timers after the sequence y. Define, for any pair of active timers i and j, the numbers:

$$M_{ij} = \max_{(r_1,r_2,\ldots,r_n)\in S_y} (r_i - r_j)$$

and, for each timer, its minimal reading in state S_Y:

$$m_i = \min_{(r_1,r_2,\ldots,r_n)\in S_y} (r_i)$$

In the mechanical representation of a state, M_{ij} is maximal (oriented) distance between the events that start timers i and j. In order to compute them, we start with the minimal position described in Observation 3, on the left of the following diagram. We then push the rod containing timer k – called, for short, timer k – to the left until the reading rod is forced to move.



Once in that position, illustrated in the right part of the diagram, the distance between the start events of timer k and any other timer i is minimized, if timer i was started before k, and maximized, if timer i was started after timer k. Thus, $M_{ki} = r_k - r_i$ for the readings corresponding to that position.

Note that in this last position, since $r_i \leq d_i$, we have that

$$r_k \leq M_{ki} + d_i,$$

but since we pushed timer k to its maximal left position, there must be at least one timer for wich $r_i = d_i$, thus we have:

Proposition 1. *The maximal reading of timer k is given by:*

$$M_k = \min\{M_{ki} + d_i\}$$

where i ranges over all active timers, including timer k.

The numbers M_{ij} give other clues about the nature of the constraints in state S_y. In the mechanical representation, two timers are *linked* if their start event are on the same rod. More formally:

Definition 4. *Two timers are linked in state S_y if the difference in their readings is constant.*

We have immediately:

Proposition 2. *Timers i and j are linked if and only if*

$$M_{ij} = -M_{ji}.$$

Finally, we say that two timers are *synchronized* if they must time out together. This can be formalized as:

Definition 5. *Timers i and j are synchronized if they are linked and if*

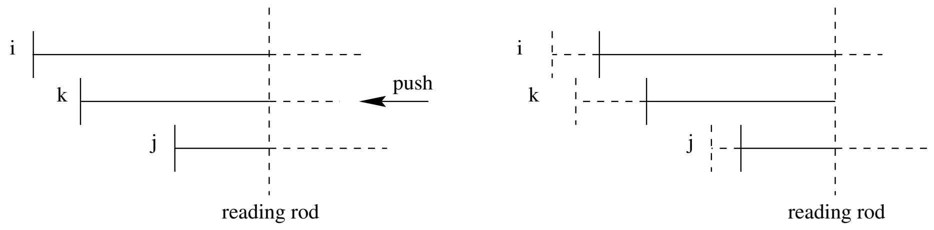
$$d_i - m_i = d_j - m_j$$

With these definitions, it is possible to give an elementary test to detect which timers can time out in a state. We have:

Theorem 1. *Timer k can time out in state S_y if and only if*

$$d_k - M_{ki} < d_i$$

for all active timers i not synchronized with k.
Proof: If timer k can timeout, then it was possible to push it completely to the left from the minimal position:



Clearly, $d_k - M_{ki} \leq d_i$ by definition of the numbers M_{ki}. If $d_k - M_{ki} = d_i$ and timer i is not synchronized with k, then timer i must time out before k since the timeout event of i and k must be separated and timer k cannot timeout before i does.

On the other hand, if $d_k - M_{ki} > d_i$, then it was possible to move timer k all the way to the left, so timer k can time out in state S_y. ■

Theorem 1 gives us the possible time out events in a state. In order to be able to predict which events are possible in a state, we must solve a last problem. Suppose that two timers can time out in a state, can they time out simultaneously? The following proposition answers this question:

Proposition 3. *If timer k and l can time out in a state S_y, then they can time out simultaneously.*
Proof: It is sufficient to show that there is at leat one position where the readings of timers k and l are respectively d_k and d_l. Starting with the minimal position, we first push timer k to the left: its reading will be d_k since timer k can time out. Then we push timer l to the left until timer k starts moving or until the reading of timer l is d_l. If the reading of timer l is d_l, we are done. If not, we get

$$M_{lk} + d_k < d_l$$

since we reached the maximal (oriented) distance between timers k and l. Thus

$$d_k < d_l - M_{lk}$$

implying that timer l could not time out in the first place. ■

Finally, we can give the complete description of possible events after a sequence y. We have:

Theorem 2. *A event e is possible in state S_y if and only if:*

a) The set of timers that are started in e is a subset of the inactive timers.

b) The set of timers that time out in e is a subset of the timers that can time out, together with their synchronized timers.

5 Computing the Next State

In this section, we will show how, given a possible event e in state S_y, then we can compute the new values of M'_{ij} and m'_i.

The first step is to compute the new values m'_i. These are obtained by *pushing* the rods that time out in event e.

Proposition 4. *Let N be the set of timers that are started in event e, and T be the set of timers that time out in event e. Then for all active timers,*

$$m'_i = \begin{cases} m_{ij} & \text{if } T \text{ is empty,} \\ \max_{k \in T}\{d_k - M_{ki}\} & \text{otherwise} \end{cases}$$

and for all $i \in N$

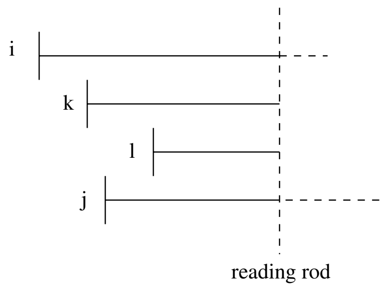
$$m'_i = 0.$$

Proof: From the minimal position, we push all the timers that time out up to their limit. This is possible by Proposition 3.

If timer i moves when we push timer k, then $r'_i = d_k - M_{ki}$. If it does not move, then $r'_i \geq d_k - M_{ki}$, thus for at least one k, $m'_i = d_k - M_{ki}$.

For new timers that are started, it is immediate that $m_i = 0$. ■

We next compute the values M'_{ij} for timers already active in S_y. Suppose, for example, that timers k and l time out in event e and consider the following diagram obtained after pushing the timers that time out in event e:



In order to compute M'_{ij} we must push timer i to the left. When we push it, either timer j will move first – in this case $M'_{ij} = M_{ij}$ –, or timer i will have reached its maximal position without touching timer j. In this case $M'_{ij} = M_i - m'_j$. We thus have the following:

Proposition 5. *If timers i and j are active in state S_y then*

$$M'_{ij} = \min\{M_{ij}, M_i - m'_j\}$$

When several timers time out and are started in event e, links will be created between events. The next operation is to *fuse* together the rods that just timed out and the new rods created. Indeed, a new timer will be linked to any timer that was linked to a timer that just timed out. These relations are captured by:

Proposition 6. *Let N be the set of timers that are started in event e, and T be the set of timers that time out in event e. Then for $i, j \in N$, and all active timers k:*

$$\begin{aligned} M'_{ij} &= M'_{ji} = 0 \\ M'_{ik} &= -m'_k \\ M'_{ki} &= \begin{cases} M_k & \textit{if } k \textit{ is not linked to any timer in } T \\ m'_k & \textit{otherwise} \end{cases} \end{aligned}$$

Proof: Clearly, if timers i and j are started together, their readings will always be equal, thus $M'_{ij} = M'_{ji} = 0$.

If timer i is started in event e, then its minimal distance from timer k will be m'_k, thus $M'_{ik} = -m'_k$.

Finally, if timer k is linked to one of the timers that times out, it will be linked to any timer that is started in e, thus $M'_{ki} = m'_k$.

Otherwise, the distance between timer k and timer i is the maximal value of timer k in state S_y. ■

6 Example of Analysis

As a simple example of constraint generation, let's consider the sequence:

$$\begin{array}{c} \\ \mathbf{T}_1 \\ \mathbf{T}_2 \\ \mathbf{T}_3 \\ \mathbf{T}_4 \\ \mathbf{T}_5 \end{array} \begin{array}{c} a \\ \begin{bmatrix} S \\ S \\ - \\ - \\ - \end{bmatrix} \end{array} \begin{array}{c} b \\ \begin{bmatrix} - \\ - \\ - \\ S \\ - \end{bmatrix} \end{array} \begin{array}{c} c \\ \begin{bmatrix} - \\ T \\ S \\ - \\ - \end{bmatrix} \end{array} \begin{array}{c} d \\ \begin{bmatrix} - \\ S \\ - \\ - \\ - \end{bmatrix} \end{array} \begin{array}{c} e \\ \begin{bmatrix} T \\ - \\ - \\ - \\ S \end{bmatrix} \end{array}$$

Assuming the delays of the timers are given by the parameters $(d_1, d_2, d_3, d_4, d_5)$, we can compute the successive values of M_{ij}, m_i and M_{ij}.

1. Event a starts timer 1 and 2. By Propositions 1 and 6, we have:

$\mathbf{M_{ij}}$	1	2	3	4	5
1	0	0	–	–	–
2	0	0	–	–	–
3	–	–	–	–	–
4	–	–	–	–	–
5	–	–	–	–	–
$\mathbf{m_j}$	0	0	–	–	–
$\mathbf{M_j}$	$\min(d_1, d_2)$	$\min(d_1, d_2)$	–	–	–

2. Event b starts timer 4, which is possible since timer 4 is inactive. By Proposition 6, and since no timer times out, $M_{1,4} = M_{2,4} = \min(d_1, d_2)$.

$\mathbf{M_{ij}}$	1	2	3	4	5
1	0	0	–	$\min(d_1, d_2)$	–
2	0	0	–	$\min(d_1, d_2)$	–
3	–	–	–	–	–
4	0	0	–	0	–
5	–	–	–	–	–
$\mathbf{m_j}$	0	0	–	0	–
$\mathbf{M_j}$	$\min(d_1, d_2)$	$\min(d_1, d_2)$	–	$\min(d_1, d_2, d_4)$	–

3. In event c, timer 2 times out and timer 3 is started. By Theorem 1, this event is possible if timer 1 is not synchronized with timer 2 and if:

$$d_2 - M_{2,1} < d_1 \Rightarrow d_2 < d_1$$
$$d_2 - M_{2,4} < d_4 \Rightarrow d_2 - min(d_1, d_2) < d_4$$

Given that $d_2 < d_1$, the second constraint asserts that $d_4 > 0$, which is trivial. So we get a first constraint for the system: $\mathbf{d_2 < d_1}$, and the new state:

$\mathbf{M_{ij}}$	1	2	3	4	5
1	0	–	d_2	d_2	–
2	–	–	–	–	–
3	$-d_2$	–	0	0	–
4	$\min(0, d_4 - d_2)$	–	$\min(d_2, d_4)$	0	–
5	–	–	–	–	–
$\mathbf{m_j}$	d_2	–	0	0	–
$\mathbf{M_j}$	M_1^c	–	M_3^c	M_4^c	–

where, by Proposition 1:

$$M_1^c = \min(d_1, d_2 + d_3, d_2 + d_4)$$
$$M_3^c = \min(d_1 - d_2, d_3, d_4)$$
$$M_4^c = \min(\min(0, d_4 - d_2) + d_1, \min(d_2, d_4) + d_3, d_4)$$

wia99@

4. Event d starts again timer 2, leading to the state:

$$\begin{array}{cccccc}
\mathbf{M_{ij}} & 1 & 2 & 3 & 4 & 5 \\
 & & & & & \\
1 & 0 & M_1^c & d_2 & d_2 & - \\
2 & -d_2 & 0 & 0 & 0 & - \\
3 & -d_2 & M_3^c & 0 & 0 & - \\
4 & \min(0, d_4 - d_2) & M_4^c & \min(d_2, d_4) & 0 & - \\
5 & - & - & - & - & - \\
 & & & & & \\
\mathbf{m_j} & d_2 & 0 & 0 & 0 & - \\
\mathbf{M_j} & M_1^c & M_2^d & M_3^c & M_4^c & -
\end{array}$$

where $M_2^d = \min(d_1 - d_2, d_2, d_3, d_4)$.

5. Finally, in event e, timer 1 must time out. This will generate the following constraints:

$$\begin{aligned}
d_1 - M_{1,2} < d_2 &\Rightarrow d_1 - M_1^c < d_2 \\
d_1 - M_{1,3} < d_3 &\Rightarrow d_1 - d_2 < d_3 \\
d_1 - M_{1,4} < d_4 &\Rightarrow d_1 - d_2 < d_4
\end{aligned}$$

Given the second and third constraint, $M_1^c = d_1$, so the first constraint is trivial, and we get the following constraints that make the sequence *abcde* possible:

$$\begin{aligned}
\mathbf{d_2} &< \mathbf{d_1} \\
\mathbf{d_1} - \mathbf{d_2} &< \mathbf{d_3} \\
\mathbf{d_1} - \mathbf{d_2} &< \mathbf{d_4}
\end{aligned}$$

References

1. R. Alur and D. Dill, *The Theory of Timed Automata*, Theoretical Computer Science, 126 (1994) 183-235.
2. A. Bergeron and A.M. Roy-Boulard, *Contraintes temporelles paramétrisées et produits synchronisés*, Proc. of NOTERE'97, Pau, France, (1997) 191-205.
3. A. Bergeron and A.M. Roy-Boulard, *Contrôleurs temps-réel paramétrisés*, Proc. of ADPM'98, Reims, France, (1998).
4. A. Bergeron, *On the Rational Behaviors of Concurrent Timers*, Theoretical Computer Science 189 (1997) 229-237.
5. B. Berthomieu and M. Diaz, *Modeling and Verification of Time Dependent systems Using timed Petri Nets*, IEEE Transactions on Software Engineering, Vol. 17, No. 3, (1991), 259-273.

Animation of the Generation and Computation of Finite Automata for Learning Software

Beatrix Braune, Stephan Diehl, Andreas Kerren, and Reinhard Wilhelm

University of Saarland, PO Box 15 11 50, D-66041 Saarbrücken, Germany
{braune, diehl, kerren, wilhelm}@cs.uni-sb.de

Abstract. In computer science methods to aid learning are very important, because abstract models are used frequently. For this conventional teaching methods do not suffice. We have developed a learning software, that helps the learner to better understand principles of compiler construction, in particular lexical analysis. The software offers on the one hand an interactive introduction to the problems of lexical analysis, in which the most important definitions and algorithms are presented in graphically appealing form. Animations show how finite automata are created from regular expressions, as well as, how finite automata work. We discuss principles used throughout the design of the software and give some preliminary results of evaluations of the software and discuss related work.

1 Introduction

The daily task of a computer science lecturer/teacher is to teach abstract knowledge and to promote the correct and lasting understanding of this knowledge by the listeners. For example, assume that a lecturer wants to describe the functionality of a pushdown automaton. In the most cases a large board and a sufficient number of colored chalk are available. Now he has the challenge to explain the functionality of the automaton on the basis of a small example input, a finite number of states and a stack picture. After three or four steps he begins to erase states in the stack picture, to add new states etc. The listener will have to spend more energy to reconstruct the complicated operational sequence of wiping and writing and to discover a sense in the disorder than to understand the functionality of a pushdown automaton. Thus the demonstration of such an automaton is difficult to reproduce by the learner. Visualization and graphic processing of the pushdown automaton are a possible solution for this dilemma. Because of dynamic processing animations are first choice for such technical problems. It is important to edit the information in such a way that cognitive and affective data processing of humans are addressed. The first is sequential and logical reasoning based on rules and regularities. The second thinks in pictures, uses analogies, ignores rules, reacts spontaneously and creatively. When we look at a suitable picture for an abstract term, we use both "information channels" and enable the connection of the actual term with a graphical imagination. This is also known as "integration learning" (see in addition [1]). Who understands to visualize information well, can increase the knowledge and understanding of learners with this method.

2 Animation of Lexical Analysis

The learning software "Animation of Lexical Analysis" has been developed with the authoring system Multimedia ToolBook 3.0 of the company Asymetrix and requires the free runtime version of ToolBook and Windows 3.x/95/98/NT4. The learning software covers the description of regular languages by regular expressions, theory of finite automata and the generation of a minimal deterministic finite automaton from any regular expression as described in [23]. Currently there is only a German version of the software.

As an introduction to lexical analysis, several animations show the fundamental components of a scanner and the cooperation between parser and scanner. Then symbols and symbol classes are explained. It is shown, how input symbols, lexical symbols, symbol classes and their internal representation are connected.

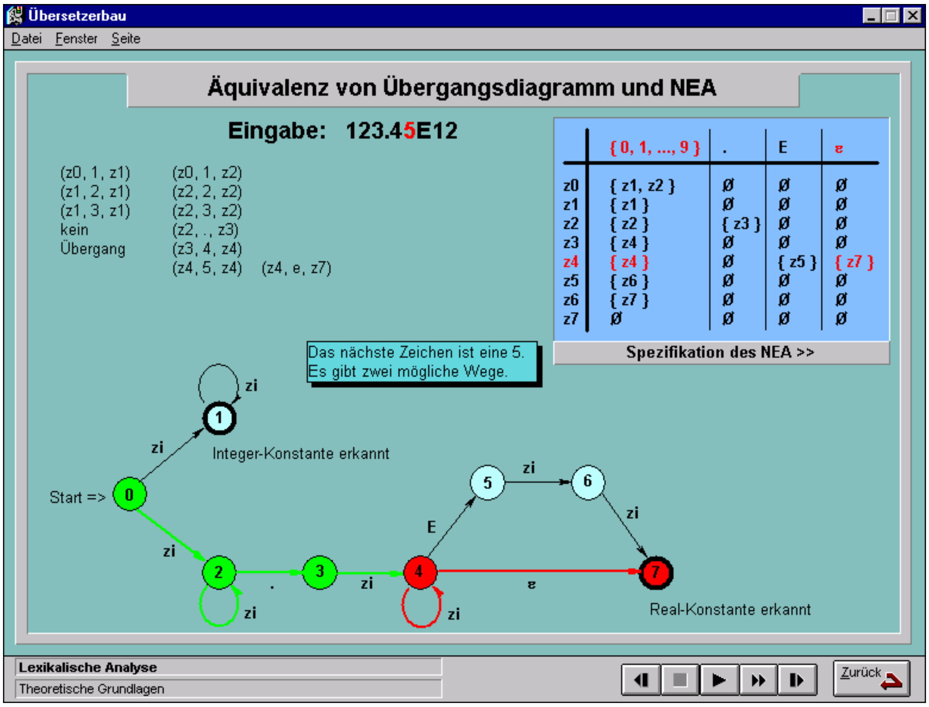
Next an overview about formal languages and an introduction to regular languages and regular expressions are given.

Then transition diagrams (TD), non-deterministic (NFA) and deterministic (DFA) finite automata are described. There are animated examples for each of these that can be controlled by the user. The equivalence between regular expressions and NFA's is explained with an fixed animated example (see Figure 1). The user can follow the parallel processing of a transition diagram and an NFA with the same input string. Currently, we see a snapshot of the NFA in state $z4$. The next character to be consumed is the character *5*. Now the NFA can read the character *5* or it can do a transition via ϵ. The animation shows both possibilities. Analogously the actual path is highlighted in the TD. The two edges from node 4 to node 7 and from node 4 back to node 4 are marked red. The shadowed box in the center of the window briefly describes what the NFA and TD actually do. In a next step, the animation will color the edge labeled *E*, update the description box, mark the state $z4$ as actual state and dismiss the second transition $(z4, \epsilon, z7)$ of the NFA, because the next character to be consumed is *E*.

Three algorithms are explained with controllable animations: the transformation of a regular expression to an NFA, the transformation of an NFA to a DFA and the transformation of a DFA to a minimal DFA.

Figure 2 shows the rules of the algorithm *regular expression to NFA* that transforms a regular expression into an NFA. In an animated example it is shown how the algorithm works. It begins with a graph consisting of two nodes and one edge that is labelled with the regular expression. Step by step the suitable rule is applied (alternative, concatenation, Kleene star or parentheses) and the graph is expanded to the resulting NFA.

In the algorithm *NFA to DFA*, the original NFA and the text of the algorithm are initially shown. With each step the corresponding line of the algorithm is



Äquivalenz von Übergangsdiagramm und NEA

Eingabe: 123.45E12

(z0, 1, z1)
(z1, 2, z1)
(z1, 3, z1)
kein Übergang

(z0, 1, z2)
(z2, 2, z2)
(z2, 3, z2)
(z2, ., z3)
(z3, 4, z4)
(z4, 5, z4) (z4, e, z7)

Das nächste Zeichen ist eine 5.
Es gibt zwei mögliche Wege.

	{ 0, 1, ..., 9 }	.	E	ε
z0	{ z1, z2 }	Ø	Ø	Ø
z1	{ z1 }	Ø	Ø	Ø
z2	{ z2 }	{ z3 }	Ø	Ø
z3	{ z4 }	Ø	Ø	Ø
z4	{ z4 }	Ø	{ z5 }	{ z7 }
z5	{ z6 }	Ø	Ø	Ø
z6	{ z7 }	Ø	Ø	Ø
z7	Ø	Ø	Ø	Ø

Fig. 1. Equivalence of Transition Diagram and NFA

highlighted and the actual nodes and edges in the graph are colored. It can be seen which nodes from the NFA build a new state in the DFA. Simultaneously to the processing of the algorithm, the new DFA is created.

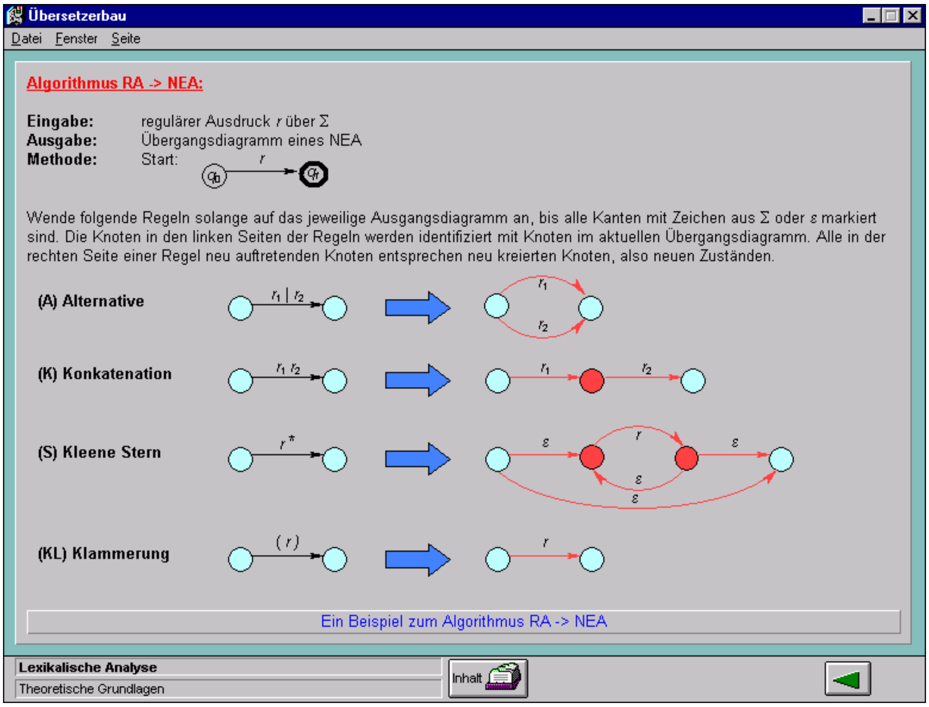
Similarly the algorithm *DFA to minimal DFA* shows the original DFA and the algorithm text. The partition classes are shown in the original graph (through coloring) and the minimal DFA is created simultaneously.

3 Design Principles

A prerequisite of implementing a good learning software is the application of good design principles. These principles were developed before the implementation of our learning software and revised during the implementation process. Some of these principles result from the research on Human-Computer-Interaction (HCI), see [17]. We propose the following guidelines:

3.1 Text

– *Font size:* If the font chosen is too small, then the user will have problems to read the text correctly, in particular sub- or superscripted text. If the font



Algorithmus RA -> NEA:

Eingabe: regulärer Ausdruck r über Σ
Ausgabe: Übergangsdiagramm eines NEA
Methode: Start:

Wende folgende Regeln solange auf das jeweilige Ausgangsdiagramm an, bis alle Kanten mit Zeichen aus Σ oder ε markiert sind. Die Knoten in den linken Seiten der Regeln werden identifiziert mit Knoten im aktuellen Übergangsdiagramm. Alle in der rechten Seite einer Regel neu auftretenden Knoten entsprechen neu kreierten Knoten, also neuen Zuständen.

(A) Alternative

(K) Konkatenation

(S) Kleene Stern

(KL) Klammerung

Ein Beispiel zum Algorithmus RA -> NEA

Lexikalische Analyse
Theoretische Grundlagen

Fig. 2. From Regular Expression to NFA

is too big, then the designer of the learning system has to solve the problem of placing enough information on the page.

– *Alignment:* Justification is not suitable for small text widths. In this case we prefer left alignment.
– *No serifs if the font is small:* Small fonts with serifs are difficult to read, because monitor resolution is not compareable with printer resolution. In computer science formulas with superscripted or subscripted letters are used frequently. This letters are very bad to read if we use a font with serifs.

3.2 Colors

– *Use few colors only:* Too many colors can irritate the user of the learning software. But colors help to direct the user's attention. Therefore colors should be used for things, to which the user's attention should be drawn.
– *Colors should harmonize:* The use of a light background color doesn't allow light font colors. The contrast between the background and the objects located on it should be high enough.
– *One fixed color for one fixed meaning:* Colors for certain links or buttons should not be changed or merged, e.g. the color blue is used for "hotword" links in our software.

3.3 Screen Arrangement

- *Main activity in one window only:* The attention of the user should be directed to one goal only. Too many windows on the screen can promote disorientation.
- *One lesson on one page if possible:* In order to avoid disorientation each basic lesson is arranged on a single page. Additional information is reached by using "hotwords" or buttons.
- *Consistent design:* Certain window areas should always be on the same place, e.g. the control buttons of the animations are consistently located in a special bar below the main window. All links must have one fixed color in case of "hotwords" or one fixed symbol in case of links to animations, definitions, etc.
- *No overloading of windows:* If a window contains very much text and many animations, then the user has difficulties to understand the important informations.

3.4 Definitions

- *Accumulate definitions:* All definitions relevant to lexical analysis are accumulated in an independent window. A first advantage is the space reduced in the main window, which is important, if the definition is very long. Furthermore the user has an overview of all definitions and can look up definitions. They can be sorted in alphabetical order or in succession of their occurence in the explanatory text.

3.5 Orientation and Navigation

- *Easy navigation and good orientation:* The actual chapter and section of the lessons are shown in a state bar located below the main window. The user can navigate to the index and from this point to another page by clicking an *Index*-button in the state bar.

3.6 Animations

- *Flexible control:* Animations should be adjustably in speed. It should be possible to execute them step by step, to stop and to reset them at each point in time. A reversed execution of animations is not in all cases meaningful and also it is frequently technically difficult to realize. However, as an alternative an Undo-operation is appropriate, that allows for a finite number of backtrack steps. Control buttons should have a well-known look, perhaps like the control buttons of a cassette recorder.
- *Clearly defined object movements:* Movements of objects should be made as directly as possible to their target, but not move over too many other objects. Several objects, that are not logical coherent, should not be moved at the same time and the movement should not be too complex and jerky.

- *Direct feedback of user actions:* Particularly in context with animations an optical feedback of the software is important, if users interact with the system. If an animation is stopped, then this stop should take place immediately and the animation should not continue a undefined amount of time.
- *Minimal memorization requirements for users:* Animations and appropriate assertions should run within a spatial and logical framework. With dynamic, automatically generated animations at runtime, this principle often conflicts with high requirement for space on the screen.
- *Spatial requirements of an animation:* More complex animations should take place on a page its own. Smaller animations should be arranged near their textual explanation.

The most challenging task when developing the learning software was to try to satisfy as much as possible of the above partially conflicting requirements.

4 Evaluation

Our target groups are students, who take a computer science course at high school, as well as students of computer science. In a pre-evaluation we left the students alone with the system. They should move independently through the graphical environment of the system and discover the learning contents on their own. With this approach we made good experiences, whereby we presupposed that the students have already been familiar with the operating system Microsoft Windows. The students got along well with the learning system, since they met a well known graphical user interface. If previous knowledge in the compiler design was available, then we noticed a better acceptance of the system, as with students, who knew still nothing about the construction of compilers. The students moved playfully through the visualizations and animations and were also able to connect these correctly with the theoretical background (definitions, algorithms, ...). Surprisingly this method worked so well that the students referred us to inconsistencies and typing errors in definitions. Students liked the optical organization of the user interface and animations.

Further presentations of our system at teachers advanced training (International Conference and Research Center for Computer Science, Dagstuhl Castle) and at the booth of the University of Saarland on the computer fair CeBIT98 and CeBIT99 in Hannover (Germany) provided positive feedback. However these measures are not yet sufficient for a serious evaluation. For this reason we cooperate at present with cognitive psychologists and develop an experiment for schools and universities, in order to receive statistically significant data of our software. For this certain aspects and characteristics of our work, e.g. the page layout or the animation control are regarded separately, while all other variable system properties remain unchanged. The use of visualization and animation is confronted to the use of the doctrine of a teacher. We still are in the preparation, so there are no results yet.

We have done the pre-evaluation with 8 highschool students (16-18 years old, Oberstufe Gymnasium) from a computer science course and got some preliminary results: 6 of them would use the system at home, 3 of them had problems with the control of the animation, none had problems with "hotwords" and only one partially disliked the design of the pages and examples.

5 Related Work

In recent years at the University of Saarland also other visualizations in the context of compiler design have been developed, including visualizations of abstract machines for imperative, logical and functional programming languages ([12], [21] and [22]). These visualizations were implemented under X-Windows (UNIX). They show the effects of the execution of machine instructions on the run time stack and heap, howewer they contain few animations. Furthermore a tool was developed for the visualization of graphs from the area of compiler design, called VCG ("Visualization of Compiler Graphs"). The VCG tool exists for several computer systems, including the Microsoft Windows system. See for this [13], [14], [15] and [16].

Another learning system developed at the University of Saarland is the "Animation of Semantical Analysis" [10], [11]. This application illustrates and animates the basic tasks of semantical analysis by textual and graphical examples. It covers basic knowledge, like the concepts of scoping and visibility, checking of context conditions (identification of identifiers, checking of type consistency), overloading of identifiers and polymorphism. The corresponding algorithms for analysis can be examined with many examples. The user can even enter his own example programs and specifications. From these inputs animations and visualizations are generated.

Also there exist a huge number of algorithm animations today, there is only a small number of fundamental work in the field. Marc H. Brown developed several algorithm animation systems, like BALSA, ZEUS, CAT, etc. These systems are frameworks, in which algorithms can be animated by annotations ("interesting events") and by definitions of graphical views ([2], [3], [4], [5] and [6]). John T. Stasko conceived the path transition paradigm and implemented it in the systems TANGO, XTANGO, SAMBA, etc., see [18], [19] and [20]. Also these systems use the concept of "interesting events". All newer versions of the above systems are complete environments, which offer some editors for the creation of views, in which the algorithms are animated. The WEB-based animation system CAT (or the newer JCAT, which is implemented in the programming language *Java*) is a complete development environment for the creation of algorithm animations in the WWW. This system offers more possibilities than ad-hoc programmed Java applets. It is possible to create algorithm animations, which a teacher can demonstrate to his students online. The interaction of the students is limited thus, but the system represents a step towards the so-called "electronic classroom". An animation can be configured in such a way that the students have the possibility to intervene interactively. They can control the animation and select other views on the algorithm. The paper [9] gives a good outline of most of the systems for algorithm animation mentioned above.

6 Conclusion

We have developed a learning software "Animation of Lexical Analysis" that helps the learner to better understand principles of compiler construction, in particular lexical analysis. The software offers on the one hand an interactive introduction to the problems of lexical analysis, in which the most important definitions and algorithms are presented in graphically appealing form. Animations show how finite automata are created from regular expressions, as well as, how finite automata work.

In our current evaluation we would like to find out whether the presentation of the learning content through the learning software has pedagogical advantages and where the software indicates weaknesses. Questions to be answered are for example, whether animations can be controlled intuitively, where the animation controls should be placed etc. From a technical point of view the use of the authoring system MTB 3.0 is questionable. It has large restrictions and the runtime system takes up much storage space. For these reasons usually important sections of the software must be implemented in another programming language, like *C*, when using authoring systems. The advantage of the system is its simplicity of operation and programming.

A new generative approach to learning software is pursued in our current project GANIMAL, that is funded by the "Deutsche Forschungsgemeinschaft – DFG". The goal of the project is to create an explorative learning software for compiler design, in which for each compiler phase the implementation **and** the appropriate visualization or animation are generated from specifications automatically. To achieve platform independence we use the programming language *Java*. Experience with designing the learning software presented here as well as its evaluations will serve as a basis for the GANIMAL project (see also [7], [8]).

The experience gained is not only applicable to the technical area of computer science, but can be transferred also to other areas, in which processes are to be visualized, for instance the medicine, electro-technology, etc. The reader finds further information about the current level of development, as well as the newest versions of the software in the WWW [24].

References

1. A. Alteneder. *Visualize with the Help of Computers: Computergraphics and Computer Animations.* Siemens, VCH (in German), 1993.
2. M. H. Brown, M. A. Najork. *Collaborative Active Textbooks: A Web-based Algorithm Animation System for an Electronic Classroom.* SRC Research Report 142, DEC, 1996.
3. M. H. Brown. *Algorithm Animation.* MIT Press, 1987.
4. M. H. Brown. *Zeus: A System for Algorithm Animation and Multi-View Editing.* SRC Research Report 75, DEC, 1992.
5. M. H. Brown. *The 1992 SRC Algorithm Animation Festival.* In IEEE Symp. on Visual Languages, pp. 116-123, 1993.
6. M. H. Brown, R. Sedgewick. *A System for Algorithm Animation.* In SIGGRAPH '84, Computer Graphics 18(3), pp. 177-186, 1984.

7. S. Diehl, T. Kunze, A. Placzek. *GANIMAM: Generation of Interactive Animations of Abstract Maschines.* In Proceedings of "Smalltalk und Java in Industrie und Ausbildung STJA'97" (in German), pp. 185-190, Erfurt (Germany), 1997.
8. S. Diehl, T. Kunze. *Visualizing Principles of Abstract Machines by Generating Interactive Animations.* In Proceedings of Workshop on Principles of Abstract Machines, Pisa (Italy), 1998.
9. A. Hausner, D. P. Dobkin. *Making Geometry Visible: an Introduction to the Animation of Geometric Algorithms.* Computer Science Department, Princeton University, 1996.
10. A. Kerren. *Animation of the Semantical Analysis.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1997.
11. A. Kerren. *Animation of the Semantical Analysis.* In Proceedings of "8. GI-Fachtagung Informatik und Schule INFOS99" (in German), pp. 108-120, Informatik aktuell, Springer, 1999.
12. G. Kohlmann. *Visualization of the abstract P-Machine.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1995.
13. I. Lemke. *Development and Implementation of a Visualization Toolkit for Applications in Compiler Construction.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1994.
14. I. Lemke, G. Sander. *Visualization of Compiler Graphs.* User Documentation, 1994.
15. G. Sander. *Visualization of Compiler Graphs.* Technical Report, University of Saarland, Saarbrücken (Germany), 1995.
16. G. Sander. *Visualization Techniques for Compiler Construction.* Dissertation (in German), University of Saarland, Saarbrücken (Germany), 1996.
17. B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* 3rd Edition, Addison-Wesley, 1997.
18. J. T. Stasko. *A Framework and System for Algorithm Animation.* Computer, 18(2), pp. 258-264, 1990.
19. J. T. Stasko. *The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces.* Journal of Visual Languages and Computing (1), pp. 213-236, 1990.
20. J. T. Stasko. *Using Student-Built Algorithm Animations as Learning Aids.* Technical Report GIT-GVU-96-19, Georgia Institute of Technology, Atlanta, 1996.
21. B. Steiner. *Visualization of the abstract Machine MaMa.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1992.
22. S. Wirtz. *Visualization of the abstract Machine WiM.* Master's Thesis (in German), University of Saarland, Saarbrücken (Germany), 1995.
23. R. Wilhelm, D. Maurer. *Compiler Design: Theory, Construction, Generation.* Addison-Wesley, 1995.
24. "http://www.cs.uni-sb.de/RW/anim/animcomp_e.html" and "http://www.cs.uni-sb.de/GANIMAL"

Metric Lexical Analysis*

Cristian S. Calude[1], Kai Salomaa[2], and Sheng Yu[3]

[1] Computer Science Department, The University of Auckland, Private Bag 92109
Auckland, New Zealand
cristian@cs.auckland.ac.nz

[2] Department of Computing and Information Science, Queen's University
Kingston, Ontario, Canada K7L 3N6
ksalomaa@cs.queensu.ca

[3] Department of Computer Science, The University of Western Ontario
London, Ontario, Canada N6A 5B7
syu@csd.uwo.ca

Abstract. We study automata-theoretic properties of distances and quasi-distances between words. We show that every additive distance is finite. We also show that every additive quasi-distance is regularity-preserving, that is, the neighborhood of any radius of a regular language with respect to an additive quasi-distance is regular. As an application we present a simple algorithm that constructs a metric (fault-tolerant) lexical analyzer for any given lexical analyzer and desired radius (fault-tolerance index).

1 Introduction

You are frustrated when you type a UNIX command incorrectly and cannot find what the correct spelling is. You may be wondering why the system does not give any suggestions on what command you might want to type. Those questions concern the concepts of distances between words and neighborhoods of languages with respect to a distance and a radius.

Much work has been done in spell checking and correction, and other online dictionary applications using various methods [5,7,8,9]. Here, we study some automata-theoretic properties of different measurements of distances between words.

Let Σ be a finite alphabet. By the *neighborhood* of a word $w \in \Sigma^*$ of radius α with respect to a distance measure δ, we mean the set of all words u that have the distance measure $\delta(u, w)$ at most α. We denote this neighborhood by $E(\{w\}, \delta, \alpha)$. Naturally, the neighborhood of a language L of a radius α with respect to δ, denoted $E(L, \delta, \alpha)$, is the union of $E(\{w\}, \delta, \alpha)$ for all words $w \in L$. A distance δ is said to be finite if $E(\{w\}, \delta, \alpha)$ is finite for all $w \in \Sigma^*$ and

* The work reported here has been supported by the Natural Sciences and Engineering Research Council of Canada Grants OGP0041630, OGP0147224 and was done during the visit of the first author to the Department of Computer Science, The University of Western Ontario, Canada.

$\alpha \geq 0$. Informally, δ is said to be additive if its measurement distributes over concatenation, and regularity-preserving if $E(R, \delta, \alpha)$ is regular for every regular language R and radius $\alpha \geq 0$.

In this paper, we prove that every additive distance is finite. We also show, as our main result, that every additive distance (or quasi-distance) is regularity-preserving. Examples of various additive and non-additive distance measures are also given in the paper.

As an application of the main result, we construct a very simple algorithm that transforms a given lexical analyzer to a metric (fault-tolerant) lexical analyzer for an arbitrary radius (fault-tolerance index).

The paper is organized as follows: In the next section we introduce the basic notation. In Section 3, we define distances and quasi-distances. Our main results concerning finite, additive, and regularity-preserving distance measures are presented in Section 4. In the last section we define metric lexical analyzers and describe a simple algorithm that constructs a metric lexical analyzer for a given lexical analyzer and desired radius.

2 Preliminaries

We assume that the reader is familiar with the basics of formal languages and finite automata in particular, cf. [4,10,11]. Here we introduce the notation we will use in the later sections.

The symbol Σ denotes a finite alphabet and Σ^* the set of finite words over Σ. The empty word is denoted by λ and the length of a word $w \in \Sigma^*$ by $|w|$. The shuffle of words $u, v \in \Sigma^*$,

$$\omega(u, v) \subseteq \Sigma^*$$

is the set of all words $x_1 y_1 x_2 \ldots x_m y_m$ such that $u = x_1 \cdots x_m$, $v = y_1 \cdots y_m$, $x_i, y_i \in \Sigma^*$, $i = 1, \ldots, m$, $m > 0$. The catenation of languages $S, T \subseteq \Sigma^*$ is denoted by ST.

A deterministic finite automaton (DFA) is a five-tuple $A = (Q, \Sigma, \gamma, s, F)$ where Q is the finite set of states, Σ is the finite alphabet, $s \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\gamma : Q \times \Sigma \to Q$ is the state-transition function. If A is defined as above except that γ is a function $Q \times \Sigma \to \mathcal{P}(Q)$ then we say that A is a nondeterministic finite automaton (NFA). (Here $\mathcal{P}(Q)$ is the set of subsets of Q.)

The state-transition relation γ of an NFA is extended in the natural way to a function $\hat{\gamma} : Q \times \Sigma^* \to \mathcal{P}(Q)$. We denote also $\hat{\gamma}$ simply by γ and the language accepted by A is $L(A) = \{w \in \Sigma^* \mid \gamma(s, w) \cap F \neq \emptyset\}$.

3 Distances and Quasi-distances

We want to measure the distance between distinct words of Σ^*. Let S be a set. We say that a function $\delta : S \times S \to [0, \infty)$ is a *distance* if it satisfies the following three conditions:

(D1) $\delta(x,y) = 0$ iff $x = y$, for all $x, y \in S$,
(D2) $\delta(x,y) = \delta(y,x)$, for all $x, y \in S$,
(D3) $\delta(x,z) \le \delta(x,y) + \delta(y,z)$, for all $x, y, z \in S$.

Condition (D3) is called the triangle-inequality. A function $\delta : S \times S \to [0, \infty)$ that satisfies (D2) and (D3) and the weaker condition

(D1') $\delta(x,x) = 0$, for all $x \in S$,

is called a *quasi-distance* on S. A quasi-distance allows the possibility that $\delta(x,y) = 0$, for $x \neq y$.

Note that if δ is a quasi-distance on S we can define an equivalence relation $\sim_\delta$ on S by setting $x \sim_\delta y$ iff $\delta(x,y) = 0$. Then the mapping δ' defined by $\delta'([x]_{\sim_\delta}, [y]_{\sim_\delta}) = \delta(x,y)$ is a distance on $S/\sim_\delta$. (Since δ satisfies the condition (D3) it follows that the value of $\delta'([x]_{\sim_\delta}, [y]_{\sim_\delta})$ does not depend on the representatives x and y.)

Let δ be a (quasi-) distance on S, $K \subseteq S$ and $\alpha \ge 0$. The *neighborhood of K of radius α* (with respect to δ) is

$$E(K, \delta, \alpha) = \{x \in S \mid (\exists y \in K)\ \delta(x,y) \le \alpha\}.$$

A natural distance between words of the same length is the so called *Hamming distance.* Since we need to compare also words of different lengths, there is more than one natural way to extend Hamming distance.

Let # be a symbol not appearing in Σ and put $\Gamma = \Sigma \cup \{\#\}$. For $a, b \in \Gamma$ define

$$\Delta(a,b) = \begin{cases} 1, & \text{if } a \neq b, \\ 0, & \text{if } a = b. \end{cases}$$

Define $\Delta_n : \Gamma^n \times \Gamma^n \to \mathbb{N}$ by setting

$$\Delta_n(x_1 \cdots x_n, y_1 \cdots y_n) = \sum_{i=1}^{n} \Delta(x_i, y_i).$$

The *prefix-Hamming distance* δ_{pH} on Σ^* is defined as follows. Let $u, v \in \Sigma^*$. Then

$$\delta_{\mathrm{pH}}(u,v) = \begin{cases} \Delta_{|v|}(u\#^k, v), & \text{if } k = |v| - |u| \ge 0, \\ \Delta_{|u|}(u, v\#^k), & \text{if } k = |u| - |v| > 0. \end{cases}$$

The prefix-Hamming distance counts the number of distinct symbols in the first $\min\{|u|, |v|\}$ positions of the words u and v and adds to the result the length of the remaining suffix. It is easy to verify that δ_{pH} satisfies the triangle-inequality and, thus, it is a distance. On the other hand, this distance is not very useful from a practical point of view because inserting or deleting one letter can change the distance of given words by an arbitrary amount (depending on the length of the words).

A better extension is the function which considers all possible ways to pad both words and then takes the minimum of the obtained distances. Let $u, v \in \Sigma^*$. Then we define

$$\delta_{\mathrm{H}}(u,v) = \min\{\Delta_k(x,y) \mid k \geq \max\{|u|,|v|\}, x \in \omega(u, \#^{k-|u|}), y \in \omega(v, \#^{k-|v|})\}. \tag{1}$$

Notice that for all $u, v \in \Sigma^*$, $\delta_H(u,v) \leq \max\{|u|,|v|\}$, and $\delta_{\mathrm{H}}(u,v) = \min\{\Delta_{|uv|}(x,y) \mid x \in \omega(u, \#^{|v|}),\ y \in \omega(v, \#^{|u|})\}$.

In general, $\delta_{\mathrm{H}}(u,v) \neq \Delta_{\max\{|u|,|v|\}}(x,y)$, for every $x \in \omega(u, \#^{\max\{|u|,|v|\}-|u|})$, $y \in \omega(v, \#^{\max\{|u|,|v|\}-|v|})$. For example, take $u = abab, v = baba$ and observe that $\omega(u, \#^0) = \{u\}, \omega(v, \#^0) = \{v\}$, $\Delta_4(u,v) = 4 > \delta_{\mathrm{H}}(u,v) = \Delta_5(u\#, \#v) = 2$.

It is convenient to look at (1) as a process. Consider changing a word into another word by means of the following three types of *edit steps* ([6]): a) *insert*—insert a character into a word, b) *delete*—delete a character from a word, c) *replace*—replace one character with a different character. Edit steps can be applied in any order. For example, to change the word *abab* into *baba* we can use rule c) (replace) four times and we get *bbab*, *baab*, *babb*, *baba*. We can be more efficient by deleting the first character of *abab* to get *bab*, then insert *a* at the end, so with only two edit steps we obtain *baba*. As we have seen below, $\delta_{\mathrm{H}}(abab, baba) = 2$; it can be obtained by first constructing the extended words *abab#* and *#baba* and then computing their Δ_5 distance. In fact, we have:

Lemma 1. *For all words u, v, $\delta_{\mathrm{H}}(u,v)$ coincides with the minimal number of edit steps necessary to change u into v.*[1]

Corollary 1. *The function δ_{H} satisfies (D1)–(D3).*

The function δ_{H} is a distance by Corollary 1; as it extends Hamming's distance it is appropriate to call it the *shuffle-Hamming distance.*

An immediate property of the shuffle-Hamming distance follows: insertions and deletions of the special symbol # do not count.

Lemma 2. *For all $u, v \in \Sigma^*$, and $i \geq 0$, $\delta_{\mathrm{H}}(u,v) = \delta_{\mathrm{H}}(\bar{u}, \bar{v})$, for all $\bar{u} \in \omega(u, \#^i), \bar{v} \in \omega(v, \#^i)$.*

Other possible distances can be obtained by varying edit steps (e.g., allowing adjacent characters in one word to be interchanged while copied to the other word) or by assigning cost functions to edit steps (e.g., capturing the idea that the cost of replacing a character is less than the combined costs of deletion and insertion). See [2] for more examples of discrete distances.

4 Neighborhoods of Regular Languages

Let L be a regular language over Σ. We are interested in the following question: Which conditions the distance δ should satisfy in order to guarantee that all the

[1] This number is called the *edit-distance* in [3], pp 325–326; it has been suggested by Ulam [12].

languages $E(L, \delta, \alpha)$, $\alpha \geq 0$, are regular? We say that a distance δ is *regularity-preserving* if $E(L, \delta, \alpha)$ is a regular language for all regular languages L and $\alpha \geq 0$.

It is fairly straightforward to construct examples of distances on Σ^* that are not regularity-preserving. Here is such an example.

Example 1. Let $\Sigma = \{a, b\}$. Construct the distance δ by

$$\delta(u,v) = \begin{cases} 0, & \text{if } u = v, \\ 1/2, & \text{if } u = a^n b^n, v = a^m b^m, \text{ for some } n, m \geq 0, n \neq m, \\ 1, & \text{otherwise,} \end{cases}$$

and notice that $E(\{ab\}, \delta, 1/2) = \{a^n b^n \mid n \geq 0\}$. □

Clearly we need to impose some additional conditions on the distance δ. Note that the distance in Example 1 has the property that for $n \geq 0$ and $\alpha \geq 1/2$, the inequality $\delta(u, a^n b^n) \leq \alpha$ has infinitely many solutions. Hence, the following finiteness requirement seems to be a suitable candidate to guarantee that a distance is regularity-preserving.

We say that a (quasi-) distance δ on Σ^* is *finite* if for all $w \in \Sigma^*$ and $\alpha \geq 0$, the set $E(\{w\}, \delta, \alpha)$ is finite.

Both the shuffle-Hamming distance and the prefix-Hamming distance considered above are clearly finite. The following example shows that finiteness of a distance δ is, unfortunately, not sufficient to guarantee that δ is regularity-preserving.

Example 2. Let $\Sigma = \{a, b, c\}$. By slightly modifying the prefix-Hamming distance δ_{pH} we construct a finite distance δ on Σ^* that is not regularity-preserving.

For $u, v \in \Sigma^*$ we define

$$\delta(u,v) = \begin{cases} 3/2, & \text{if } u = a^n b a^n, v = a^n c a^n, n \geq 0, \text{ or vice versa,} \\ \delta_{\mathrm{pH}}(u,v), & \text{otherwise.} \end{cases}$$

Clearly δ satisfies the conditions (D1) and (D2), so in order to show that it is a distance it is sufficient to verify the triangle-inequality. Assuming that (D3) does not hold, we must have $x, y, z \in \Sigma^*$ such that

$$\delta(x,z) > \delta(x,y) + \delta(y,z). \tag{2}$$

Since for all $u, v \in \Sigma^*$, $\delta(u,v) \geq \delta_{\mathrm{pH}}(u,v)$ and δ_{pH} is a distance, it follows that if (2) holds, then necessarily $\delta(x,z) \neq \delta_{\mathrm{pH}}(x,z)$, that is, $x = a^n b a^n$, $z = a^n c a^n$, $n \geq 0$, or vice versa. Thus $\delta(x,z) = 3/2$, and (2) implies that $\delta(x,y) = 0$ or $\delta(y,z) = 0$. Both possibilities directly yield a contradiction.

Also, δ is finite since for any $\alpha \geq 2$ and $w \in \Sigma^*$ we have $E(\{w\}, \delta, \alpha) = E(\{w\}, \delta_{\mathrm{pH}}, \alpha)$.

To see that δ is not regularity-preserving choose $L = a^* b a^*$. Then

$$E(L, \delta, 3/2) - E(L, \delta, 1) = \{a^n c a^n \mid n \geq 0\},$$

which implies that at least one of the languages $E(L, \delta, 3/2)$ and $E(L, \delta, 1)$ is not regular. □

The above example shows that we need to look for stronger restrictions for regularity-preserving distances. Since elements of Σ^* have a unique decomposition into subwords (of given length) it is perhaps reasonable to assume that the distances should "respect" such decompositions. Thus we say that a (quasi-) distance δ on Σ^* is *additive* if always when $w = w_1 w_2$ ($w_1, w_2 \in \Sigma^*$) we have for all $\alpha \geq 0$,

$$E(\{w\}, \delta, \alpha) = \bigcup_{\beta_1+\beta_2=\alpha} E(\{w_1\}, \delta, \beta_1) E(\{w_2\}, \delta, \beta_2). \tag{3}$$

First we observe that an additive distance is always finite. Note that an additive quasi-distance δ need not be finite. If, for some $b \in \Sigma$, $\delta(b, \lambda) = 0$, then any δ-neighborhood is necessarily infinite.

Lemma 3. *Every additive distance is finite.*

Proof. Let δ be an additive distance on Σ^*. By (3), for any $w = b_1 \cdots b_k$, $b_i \in \Sigma$, $i = 1, \ldots, k$, $E(\{w\}, \delta, \alpha)$ is contained in the catenation of the languages $E(\{b_1\}, \delta, \alpha)$, …, $E(\{b_k\}, \delta, \alpha)$. Thus, it is sufficient to show that $E(\{b\}, \delta, \alpha)$ is finite for $b \in \Sigma$ and $\alpha \geq 0$.

Let $u = c_1 \cdots c_m$, $c_i \in \Sigma$, be an arbitrary word of Σ^*. The additivity condition implies that $u \in E(\{b\}, \delta, \alpha)$ iff there exists $i \in \{1, \ldots, m\}$ such that

$$\delta(b, c_i) + \sum_{j \in \{1,\ldots,m\},\ j \neq i} \delta(\lambda, c_j) \leq \alpha. \tag{4}$$

There exist only a finite number of words $u = c_1 \cdots c_m$ that satisfy the above inequality. □

Both the prefix-Hamming distance and the shuffle-Hamming distance are additive.

Proposition 1. *The distances δ_{pH} and δ_{H} defined on an alphabet Σ are additive.*

Proof. We show that δ_{H} is additive as the proof for the distance δ_{pH} is simpler.

Let $w = w_1 w_2$ be an arbitrary decomposition of a word $w \in \Sigma^*$. We show that for every $u \in \Sigma^*$,

$$u \in E(\{w_1 w_2\}, \delta_{\mathrm{H}}, \alpha) \text{ iff } u \in \bigcup_{\beta_1+\beta_2=\alpha} E(\{w_1\}, \delta_{\mathrm{H}}, \beta_1) E(\{w_2\}, \delta_{\mathrm{H}}, \beta_2).$$

Assume $\delta_{\mathrm{H}}(u, w_1 w_2) \leq \alpha$. As edit steps (in the process of changing a word into another word) can be applied in any order, we can start the process of changing u into $w_1 w_2$ in such a way to obtain first w_1 from a prefix u_1 of u, and then w_2 (from the remaining suffix u_2 of u). Consequently, $\delta_{\mathrm{H}}(u_1, w_1) + \delta_{\mathrm{H}}(u_2, w_2) = \delta_{\mathrm{H}}(u, w_1 w_2) \leq \alpha$. Conversely, if $u_i \in E(\{w_i\}, \delta_{\mathrm{H}}, \beta_i)$, $i = 1, 2$, $\beta_1 + \beta_2 \leq \alpha$, then we have $\delta_{\mathrm{H}}(u_1 u_2, w_1 w_2) \leq \delta_{\mathrm{H}}(u_1, w_1) + \delta_{\mathrm{H}}(u_2, w_2) \leq \alpha$. □

From Example 2 we know that a finite distance need not preserve regularity. Below we show that, on the other hand, additivity is a sufficient condition to guarantee that even a quasi-distance preserves regularity. Note that, as observed above, an additive quasi-distance need not be finite. First we prove the following lemma.

Lemma 4. *Assume that δ is an additive quasi-distance on Σ^*.*

(i) *For each $b \in \Sigma$ and $\alpha \geq 0$, $E(b, \delta, \alpha)$ is regular.*
(ii) *Let $b \in \Sigma$ and $\alpha \geq 0$ be fixed. There exists an integer k and numbers $0 = \alpha_1 < \ldots < \alpha_k = \alpha$ such that*

$$E(b, \delta, \alpha_i), \quad i = 1, \ldots, k,$$

are all the distinct neighborhoods of b having radius at most α.

Proof. (i) Let $u = c_1 \cdots c_m$, $m \geq 0$, $c_i \in \Sigma$, $i = 1, \ldots, m$. As in the proof of Lemma 3 it follows that $u \in E(b, \delta, \alpha)$ iff the inequality (4) holds. (Note that, in contrast to Lemma 3, δ is now only a quasi-distance, so this does not imply the finiteness of the neighborhood.)

Denote

$$\Theta = \{d \in \Sigma \mid \delta(d, \lambda) = 0\}.$$

Let Ψ be the set of finite multisets of elements of Σ,

$$\{c_i, c_{j_1}, \ldots, c_{j_r}\}$$

such that $\delta(\lambda, c_{j_l}) \not= 0 l = 1, \ldots, r$ and

$$\delta(b, c_i) + \sum_{l=1}^{r} \delta(\lambda, c_{j_l}) \leq \alpha.$$

Then $u = c_1 \cdots c_m$ satisfies the inequality (4) iff u is the shuffle of a sequence obtained by listing the elements of a multiset belonging to Ψ (in arbitrary order) and a word in Θ^*. The shuffle of a finite language and a regular language is always regular.

(ii) In the construction above the elements of the multisets belonging to Ψ completely determine the neighborhoods of radius at most α around b. Thus as the radii α_s, $s = 1, \ldots, k$, we can simply take all the (distinct) sums $\delta(b, c_i) + \sum_{l=1}^{r} \delta(\lambda, c_{j_l})$ where the multiset $\{c_i, c_{j_1}, \ldots, c_{j_r}\}$ belongs to Ψ. (Note that Ψ is a finite collection of multisets.) □

The above construction implies that Lemma 4 (ii) can be written in the following stronger form:

Corollary 2. *Assume that δ is an additive quasi-distance on Σ^* and let $b \in \Sigma$ and $\alpha \geq 0$ be fixed. Then we can write*

$$E(b, \delta, \alpha) = R_1 \cup \ldots \cup R_k$$

where $R_i = \{w \in \Sigma^* \mid \delta(b, w) = \alpha_i\}$, $i = 1, \ldots, k$, is regular.

Proof. Without loss of generality we can assume that the numbers α_i in Lemma 4 (ii) are chosen so that there exists $w_i \in \Sigma^*$ with $\delta(b, w_i) = \alpha_i$, $i = 1, \ldots, k$. Let R_i, $i = 1, \ldots, k$, be as above. By Lemma 4 (ii), $R_i = E(b, \delta, \alpha_i) - E(b, \delta, \alpha_{i-1})$, $i = 2, \ldots, k$, and $R_1 = E(b, \delta, 0)$. By Lemma 4 (i), these sets are regular. □

Now we are ready to prove the main result of this section.

Theorem 1. *Assume that δ is an additive quasi-distance on Σ^* and let $L \subseteq \Sigma^*$ be regular. Then $E(L, \delta, \alpha)$ is regular for all $\alpha \geq 0$.*

Proof. Let $\alpha \geq 0$ be fixed and let $A = (Q, \Sigma, \gamma, s, F)$ be a DFA such that $L = L(A)$. Without loss of generality we can assume that the initial state s is not reachable from any other state.

By Corollary 2, for each $b \in \Sigma$ we can write

$$E(b, \delta, \alpha) = R_1^b \cup \ldots \cup R_{k(b)}^b,$$

where

$$R_j^b = \{w \in \Sigma^* \mid \delta(w, b) = \alpha_j^b\}, \quad 0 \leq \alpha_j^b \leq \alpha,$$

is regular, $j = 1, \ldots, k(b)$. Denote $D' = \{\alpha_j^b \mid b \in \Sigma, 1 \leq j \leq k(b)\}$ and

$$D = \{\beta \leq \alpha \mid \beta = \beta_1 + \ldots + \beta_r, \beta_i \in D', 1 \leq i \leq r\}.$$

We construct an NFA $B = (Q_B, \Sigma, \gamma_B, s_B, F_B)$ such that

$$L(B) = E(L(A), \delta, \alpha).$$

Choose $Q_B = Q \times D$, $s_B = (s, 0)$ and

$$F_B = \begin{cases} F \times D \cup \{s_B\} & \text{if } \lambda \in E(L(A), \delta, \alpha) \\ F \times D & \text{otherwise.} \end{cases}$$

The transition relation γ_B is defined as follows. Let $q \in Q$, $\beta \in D$ and $b \in \Sigma$. Then

$$(q', \beta + \alpha_j^b) \in \gamma_B((q, \beta), b) \tag{5}$$

for every $q' \in \gamma(q, R_j^b)$, $1 \leq j \leq k(b)$, such that $\beta + \alpha_j^b \leq \alpha$. (Here $\gamma(q, R_j^b) = \{\gamma(q, v) \mid v \in R_j^b\}$.) Since R_j^b is regular, the set $\gamma(q, R_j^b)$ ($\subseteq Q$) can even be effectively determined.

Let $w = b_1 \cdots b_m$, $m \geq 1$, $b_i \in \Sigma$, $i = 1, \ldots, m$. Since δ is additive

$$\begin{aligned} & w \in E(L(A), \delta, \alpha) \text{ iff } (\exists u \in L(A)) \text{ such that} \\ & u \in \bigcup_{\beta_1 + \ldots + \beta_m = \alpha} E(b_1, \delta, \beta_1) \cdots E(b_m, \delta, \beta_m). \end{aligned} \tag{6}$$

In the transitions (5), on input b the first component of the states of B simulates the computation of A on an arbitrary (nondeterministically chosen) word of $v \in R_j^b$, and in the second component we correspondingly increment the distance by $\alpha_j^b = \delta(b, v)$. By observation (6), some sequence of the nondeterministic choices on input $w = b_1 \cdots b_m$ leads to an accepting state of F_B iff w is in $E(L(A), \delta, \alpha)$. By the choice of the set F_B, the NFA B accepts λ if and only if $\lambda \in E(L(A), \delta, \alpha)$. □

5 A Metric Lexical Analyzer

The major difference between a lexical analyzer and a (traditionally-defined) finite automaton is that, in a lexical analyzer, each final state is linked to an action (or a set of actions). Because of this difference, the algorithms that are designed for finite automata may not directly apply to lexical analyzers. The equivalence of two final states in a deterministic lexical analyzer requires that not only the states are equivalent in the sense of a DFA, but also that they have the same action (or actions).

There are many other features which are associated with certain types of lexical analyzers. For example, some lexical analyzers assume that each input word has an end-of-word symbol. Also, many practical lexical analyzers are implemented using a data structure called *trie* [1]. However, those features are not considered to be common or essential to general lexical analyzers.

A lexical analyzer can be considered as a special type of Moore machine [4] with all nonfinal states having the empty output (action).

For notational convenience, we formally define a lexical analyzer to be a 7-tuple

$$A = (Q, \Sigma, \Gamma, \gamma, s, F, \tau)$$

where $(Q, \Sigma, \gamma, s, F)$ is a finite automaton; Γ is a set of actions; and, $\tau : F \to \Gamma$ is an action-function. Whether A is deterministic or nondeterministic depends on whether its underlying finite automaton is a DFA or an NFA.

Let $A = (Q, \Sigma, \Gamma, \gamma, s, F, \tau)$ be a deterministic (nondeterministic) lexical analyzer. Denote by $L(A)$ the set of all words recognized by the underlying finite automaton $(Q, \Sigma, \gamma, s, F)$. For $w \in L(A)$, denote by $\hat{\tau}(w)$ the action $\tau(\hat{\gamma}(s, w))$ (the set of actions $\{\tau(f) \mid f \in F \cap \hat{\gamma}(s, w)\}$). We simply write $\tau(w)$ instead of $\hat{\tau}(w)$ if there is no confusion. The above definition implies that if $w \in L(A)$ and the (an) accepting path for w goes through several final states, only the action associated to the last final state is activated.

A simple lexical analyzer A_u is shown in Figure 1.

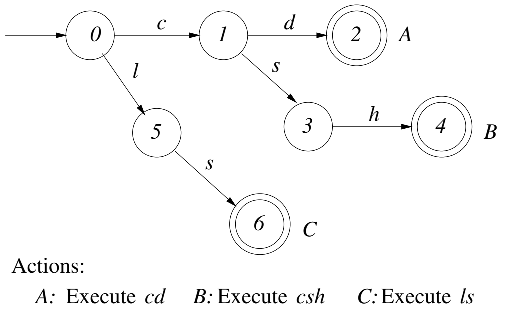


Fig. 1. A lexical analyzer A_u

Let $A = (Q, \Sigma, \Gamma, \gamma, s, F, \tau)$ be a lexical analyzer, δ a regularity-preserving distance and $\alpha \geq 0$ a radius. A lexical analyzer $A' = (Q', \Sigma, \Gamma', \gamma', s', F', \tau')$ is called a *metric lexical analyzer* of A with respect to δ and α if

(M1) $L(A') = E(L(A), \delta, \alpha)$, and
(M2) for each $w \in L(A)$, $\tau'(w) = \tau(w)$.

The proof of Theorem 1 gives a general guideline for constructing a metric lexical analyzer from a given lexical analyzer, an additive quasi-distance, and a radius. In the following, however, we consider only the shuffle-Hamming distance δ_{H}. The idea below can be easily generalized to all additive quasi-distances.

Construction: A Deterministic Metric Lexical Analyzer

Given: A deterministic lexical analyzer (DLA) $A = (Q, \Sigma, \Gamma, \gamma, s, F, \tau)$ and an integer $k > 0$.
Result: A deterministic metric lexical analyzer (DMLA)

$$A' = (Q', \Sigma, \Gamma', \gamma', s', F', \tau')$$

of A with radius k.

Construction steps:

i) Construct a nondeterministic lexical analyzer (NLA)

$$A'' = (Q'', \Sigma, \Gamma'', \gamma'', s'', F'', \tau'')$$

such that
$Q'' = Q \times \{0, \ldots, k\}$,
$\Gamma'' = \Gamma \cup \{\tilde{e} \mid e \in \Gamma\}$,
$s'' = (s, 0)$,
$F'' = \{(f, i) \mid f \in F \;\&\; i = 0, \ldots, k\}$,

$$\gamma'' : \begin{cases} (q, i) \in \gamma''((p, i), a), & \text{for } i = 0, \ldots, k, \text{ if } q = \gamma(p, a), \\ (q, i+1) \in \gamma''((p, i), b), & \text{for } i < k \text{ and } b \in \Sigma, b \neq a \text{ if } q = \gamma(p, a), \\ (q, i+1) \in \gamma''((q, i), a), & \text{for all } a \in \Sigma \text{ and } (q, i) \in Q'' \text{ where } i < k, \\ (q, i+1) \in \gamma''((p, i), \lambda), & \text{for } i < k \text{ if } q = \gamma(p, a), \text{ for some } a \in \Sigma, \end{cases}$$

$$\tau'' : \begin{cases} \tau''((f, 0)) = \tau(f), & \text{for } f \in F, \\ \tau''((f, i)) = \tilde{e}, & \text{for } f \in F \text{ and } i = 1, \ldots, k, \text{ if } \tau(f) = e. \end{cases}$$

ii) Reduce A'' by deleting those states that are not reachable from s'' or that cannot reach a final state.

iii) Construct A' using the subset construction method [4] such that Q', γ', s', and F' are defined as in a standard subset construction, except that if a new state $r \in Q'$ contains both (q, i) and (q, j) for some $q \in Q$ and $i < j$ then delete (q, j) from r; $\Gamma' \subseteq \mathcal{P}(\Gamma'')$, and $\tau'(f') = \tau(f)$ if $(f, 0) \in f'$, for some $(f, 0) \in F''$, or $\tau'(f') = \{\tau''(f'') \mid f'' \in f' \;\&\; f'' \in F''\}$, otherwise.

iv) Simplify A' by merging all the equivalent states (that have the same actions if they are final states).

Note that the above construction uses two copies of the original set of actions ($\Gamma \cup \{\tilde{e} \mid e \in \Gamma\}$) in order to guarantee that the property (M2) holds.

A nondeterministic metric lexical analyzer A''_u of A_u with radius 1 is constructed following Step i) and Step ii) and shown in Figure 2, where $\Gamma'' = \{A, B, C, \tilde{A}, \tilde{B}, \tilde{C}\}$ and $\tau''((2,0)) = A$, $\tau''((2,1)) = \tilde{A}$, $\tau''((4,0)) = B$, $\tau''((4,1)) = \tilde{B}$, $\tau''((6,0)) = C$, $\tau''((6,1)) = \tilde{C}$. We use $[\hat{\ } a_1 \cdots a_t]$ to denote all letters in $\Sigma - \{a_1, \ldots, a_t\}$, and $\cdot$ to denote all letters in Σ.

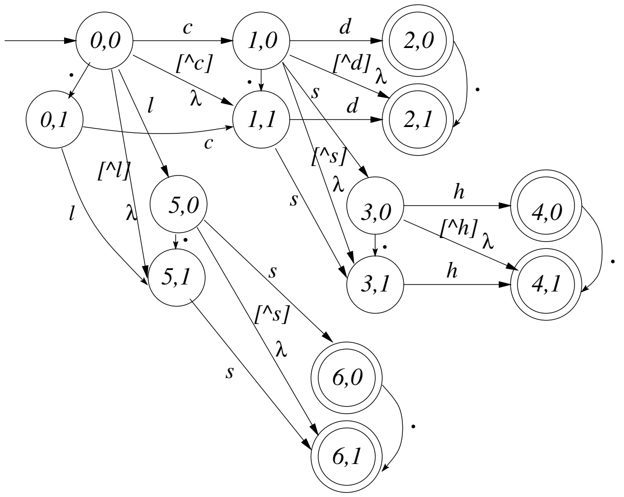


Fig. 2. A nondeterministic metric lexical analyzer A''_u

The resulting DMLA A' for A is shown in the following table, where $t1$, …, $t6$ are terminating states which have no transitions:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
c	1	5	t1	X	10	X	X	X	t2	X	X	X	X	16	t5	X	X	18	X	X	18	18
d	17	2	t1	X	9	t1	t1	t1	t2	X	X	X	X	16	t5	X	t1	t1	t1	X	t1	t1
h	21	6	t3	t2	8	t2	t2	t2	t2	t2	t2	t2	t2	t4	t4	X	X	X	X	X	t2	X
l	13	7	t1	X	10	X	X	X	t2	X	X	X	X	15	t5	X	X	19	X	X	19	19
s	20	4	3	X	10	12	12	11	t2	X	X	X	X	14	t5	t5	12	11	12	t5	11	11
ˆ	21	5	t1	X	10	X	X	X	t2	X	X	X	X	t5	t5	X	X	X	X	X	X	X
τ'			A	$\tilde{A}$	$\tilde{A}$ $\tilde{B}$ $\tilde{C}$	$\tilde{A}$	$\tilde{A}$ $\tilde{B}$	$\tilde{A}$	B	$\tilde{A}$ $\tilde{B}$	$\tilde{B}$	$\tilde{C}$		$\tilde{A}$ $\tilde{C}$	C	$\tilde{C}$	$\tilde{C}$	$\tilde{A}$			$\tilde{C}$	

$$\tau'(t1) = \{\tilde{A}\},\ \tau'(t2) = \{\tilde{B}\},\ \tau'(t3) = \{\tilde{A}, \tilde{B}\},\ \tau'(t4) = \{\tilde{B}, \tilde{C}\},\ \tau'(t5) = \{\tilde{C}\},\\ \tau'(t6) = \{\tilde{A}, \tilde{C}\}.$$

Acknowledgment. We wish to thank Dr. Kaizhong Zhang for helpful discussions and suggestions.

References

1. A.V. Aho, R. Sethi, J.D. Ullman: *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1988.
2. C.S. Calude, E. Calude: On some discrete metrics, *Bull. Math. Soc. Sci. Math. R. S. Roumanie (N. S.)* 27 (75) (1983), 213–216.
3. T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms,* MIT Press, Cambridge, MA, 1990.
4. J.E. Hopcroft, J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading, MA, 1979.
5. R.L. Kashyap, B.J. Oommen: Spelling correction using probabilistic methods, *Pattern Recognition Letters* 2, 3 (1984), 147–154.
6. U. Manber: *Introduction to Algorithms—A Creative Approach*, Addison-Wesley, Reading, MA, 1989.
7. D.J. Margoliash: *CSpell—A Bloom Filter-Based Spelling Correction Program*, Masters Thesis, Dept. of Computer Science, Univ. of Western Ontario, London, Canada, 1987.
8. A. Mateescu, A. Salomaa, K. Salomaa, S. Yu: Lexical analysis with a simple finite-fuzzy-automaton model, *Journal of Universal Computer Science*, 1, 5 (1995), 292–311.
9. M. Mor, A.S. Fraenkel: A hash code method for detecting and correcting spelling errors, *Comm. ACM* 25, 12 (1982), 935–938.
10. A. Salomaa: *Formal Languages,* Academic Press, New York, 1973.
11. S. Yu: Regular languages. In: *Handbook of Formal Languages, Vol. I.* (G. Rozenberg, A. Salomaa, eds.) pp. 41–110, Springer-Verlag, Berlin, 1997.
12. S. Ulam: Some ideas and prospects in biomathematics, *The Annual Review of Biophysics and Bioengineering*, 1 (1972), 277–292. Reprinted in S. Ulam: *Science, Computers, and People* (M. C. Reynolds, G.- C. Rota, eds.), Birkhäuser, Boston, 1986, 115-136.

State Complexity of Basic Operations on Finite Languages[⋆]

C. Câmpeanu[1], K. Culik II[2], Kai Salomaa[3], and Sheng Yu[3]

[1] Fundamentals of Computer Science Department, Faculty of Mathematics
University of Bucharest, Romania cezar@funinf.math.unibuc.ro
[2] Department of Computer Science, University of South Carolina
Columbia, SC 29208, USA culik@cs.scarolina.edu
[3] Department of Computer Science, The University of Western Ontario
London, Ontario, Canada N6A 5B7 {ksalomaa, syu}@csd.uwo.ca

Abstract. The state complexity of basic operations on regular languages has been studied in [9,10,11]. Here we focus on finite languages. We show that the catenation of two finite languages accepted by an m-state and an n-state DFA, respectively, with $m > n$ is accepted by a DFA of $(m - n + 3)2^{n-2} - 1$ states in the two-letter alphabet case, and this bound is shown to be reachable. We also show that the tight upper-bounds for the number of states of a DFA that accepts the star of an n-state finite language is $2^{n-3} + 2^{n-4}$ in the two-letter alphabet case. The same bound for reversal is $3 \cdot 2^{p-1} - 1$ when n is even and $2^p - 1$ when n is odd. Results for alphabets of an arbitrary size are also obtained. These upper-bounds for finite languages are strictly lower than the corresponding ones for general regular languages.

1 Introduction

Many applications of regular languages use essentially finite languages. In [9, 10,11], the state complexity of basic operations on regular languages has been studied. It is interesting and important to know whether those state-complexity results still hold for finite languages. For example, $(2m - 1)2^{n-1}$ is the number of states of a minimal DFA, in the worst case, that accepts the catenation of an m-state and an n-state DFA language. Does the catenation of two DFA, each accepting a finite language, need the same number of states in the worst case? May it be significantly smaller?

It is known [4] that a minimal DFA that accepts the reversal of an n-state DFA language needs 2^n states in the worst case. This fact determines that Brzozowski's DFA minimization algorithm [1,7], which uses two reversals, is exponential in time and space in the worst case. However, this algorithm is faster than other algorithms in many experiments. It is a natural question whether this algorithm has a polynomial time or space complexity in the case of finite

[⋆] The work reported here has been supported by the Natural Sciences and Engineering Research Council of Canada Grants OGP0041630 and OGP0147224.

languages. This question is very much related to the state complexity of the reversal of finite languages.

In this paper, we focus on the above mentioned problems and on the state complexity of basic operations on finite languages, in general. We show that for an n-state DFA A accepting a finite language L, a minimal DFA that accepts L^* has $2^{n-3}+2^{n-t-2}$ states in the worst case, where $t \geq 2$ is the number of final states in A (except the starting state). Note that for $t=1$, this bound is simply $n-1$.

For the catenations of finite languages, we show that a minimal DFA that accepts the catenation of two finite languages, which are accepted by an m-state DFA and an n-state DFA, respectively, has at most

$$\sum_{i=0}^{m-2} \min\left\{k^i, \binom{n-2}{\leq i}, \binom{n-2}{\leq t-1}\right\} + \min\left\{k^{m-1}, \binom{n-2}{\leq t}\right\}$$

states, where k is the size of the alphabet and t is the number of final states in the first automaton. Notice that this bound depends very much on t. If t is a constant, then this bound is $O(mn^{t-1}+n^t)$, which is polynomial. In particular, when $t=1$, it is $m+n-2$. In the case of a two-letter alphabet (with an arbitrary t), this bound is $(m-n+3)2^{n-2}-1$. We give examples to show that this bound is reachable.

We also show that $\sum_{i=0}^{t-1} k^i + 2^{n-1-t}$ is an upper bound on the number of states for a minimal DFA that accepts the reversal of a finite language accepted by an n-state DFA, where t is the smallest integer such that $2^{n-1-t} \leq k^t$. This bound is, in the case of a two-letter alphabet, $3 \cdot 2^{p-1}-1$ if $n=2p$ or 2^p-1 if $n=2p-1$. We also give examples to show that the latter bounds are reachable. Unfortunately, these results show that Brzozowski's DFA minimization algorithm is still exponential in the worst case even for finite languages.

We also consider the state complexity of operations on finite languages in the case of a one-letter alphabet.

2 Preliminaries

Let T be a finite set. Denote by $\#T$ the cardinality of T and by T^* the free monoid generated by T. The empty word, i.e., the neutral element of T^*, is denoted by λ and $T^+ = T^* - \{\lambda\}$. For $w \in T^*$, denote by $|w|$ the length of w. We define

$$T^l = \{w \in T^* \mid |w| = l\},\ \ T^{\leq l} = \bigcup_{i=0}^{l} T^i,\ \text{and } T^{<l} = \bigcup_{i=0}^{l-1} T^i.$$

If $T=\{t_1,\ldots,t_k\}$ is an ordered set, $k>0$, the lexicographical order on T^*, denoted $\preceq_l$, is defined by: $x \preceq_l y$ iff $x=y$ or $|x|<|y|$ or $|x|=|y|$ and $x=zt_iv$, $y=zt_ju$, $i<j$, for some $z,u,v \in T^*$ and $1 \leq i,j \leq k$. We say that x is a prefix of y, denoted $x \preceq_p y$ if $y=xz$ for some $z \in T^*$. The relation $\preceq_p$ is a partial order on T^*.

A deterministic finite automaton (DFA) is a quintuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is the finite nonempty set of states; Σ is the finite nonempty alphabet; $q_0 \in Q$ is the starting state; $F \subseteq Q$ is the set of final states; and $\delta : Q \times \Sigma \longrightarrow Q$ is the transition function. We extend δ from $Q \times \Sigma$ to $Q \times \Sigma^*$ by $\overline{\delta}(q, aw) = \overline{\delta}(\delta(q, a), w)$ and $\overline{\delta}(q, \lambda) = q$ for $q \in Q$, $a \in \Sigma$, and $w \in \Sigma^*$. We usually denote $\overline{\delta}$ by δ if there is no confusion.

The language recognized by the automaton A is $L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$. Two automata are equivalent if they recognize the same language.

For simplicity, in what follows, we assume that $Q = \{0, 1, \ldots, \#Q - 1\}$ and $q_0 = 0$. We also assume that δ is a total function, i.e., that the automaton is complete.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Then

a) a state s is said to be accessible if there exists $w \in \Sigma^*$ such that $\delta(0, w) = s$;

b) a state s is said to be useful if there exists $w \in \Sigma^*$ such that $\delta(s, w) \in F$.

It is clear that for every DFA A there exists an automaton A' such that $L(A') = L(A)$ and every state of A' is accessible and at most one state is useless (the sink state). The DFA A' is called a *reduced* DFA. We will use only reduced DFA in the following.

A DFA $A = (\Sigma, Q, q_0, \delta, F)$ is said to be *minimal* if for every other automaton $A' = (\Sigma, Q', q'_0, \delta', F')$ such that $L(A) = L(A')$, we have $\#Q \leq \#Q'$.

A minimal DFA has at most one useless state.

Let $L \subseteq \Sigma^*$ and $x, y \in \Sigma^*$. Then $x \equiv_L y$ if for all $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$. Clearly, $\equiv_L$ is an equivalence relation on Σ^*. The number of states in a minimal DFA that accepts L is exactly the number of equivalence classes of $\equiv_L$ [3]. If $L = L(A)$ and p, q are states of the DFA $A = (\Sigma, Q, q_0, \delta, F)$ we denote also $p \equiv_L q$ (or simply $p \equiv q$) if for all $z \in \Sigma^*$, $\delta(p, z) \in F$ iff $\delta(q, z) \in F$.

For basic definitions and results in automata theory, the reader may refer to [5,3,11].

3 Star Operation on Finite Languages

In [9] (also in [11]), it was shown that for any n-state (complete) DFA A, there exists a minimal DFA of at most $2^{n-1}+2^{n-2}$ states that accepts $L(A)^*$. Examples were also given to show that this bound is reachable. In this section, we show that in the case that A accepts a finite language rather than an infinite regular language, the corresponding bound is exactly $2^{n-3} + 2^{n-4}$. The latter is exactly one-fourth of the former.

Let A be an n-state DFA accepting a finite language. If A has only one final state, it is clear that a minimal DFA accepting $L(A)^*$ needs at most $n-1$ states. Note that this is not true in general for an n-state DFA accepting an infinite regular language. It has been shown that the upper bound $2^{n-1} + 2^{n-2}$ can be reached even for n-state DFA with only one final state.

In the following, we consider DFA with at least two final states.

Theorem 1. *Let $A = (Q, \Sigma, 0, \delta, F)$ be a DFA accepting a finite language L, where $0 \notin F$, $\#F = t \geq 2$, $\#Q = n \geq 4$. Then there exists a DFA of at most $2^{n-3} + 2^{n-t-2}$ states that accepts L^*.*

Proof. We first construct an NFA A' from A by adding a λ-transition from each final state $f \in F$ to 0. Formally, $A' = (Q, \Sigma, \delta', 0, F)$ where $\delta' : Q \times \Sigma \to 2^Q$ is defined for each $p \in Q$ and $a \in \Sigma$ as follows:

$$\delta'(p, a) = \begin{cases} \{q\} & \text{if } q = \delta(p, a) \text{ and } q \notin F, \\ \{q, 0\} & \text{if } q = \delta(p, a) \text{ and } q \in F. \end{cases}$$

Clearly, A' accepts $L(A)^+$.

Next we construct a DFA $B = (Q_B, \Sigma, \delta_B, 0_B, F_B)$ from A' using the standard subset-construction method [3,8] and, furthermore, make the starting state of B a final state which guarantees that $L(B) = L(A)^*$. Then we have $Q_B \subseteq 2^Q$, $0_B = \{0\}$, $F_B = \{P \in Q_B \mid P \cap F \neq \emptyset\} \cup \{0_B\}$, and $\delta_B(P, a) = \cup_{p \in P} \delta'(p, a)$.

In the following, we assume that, in A, $(n-1)$ is the sink state and $(n-2)$ is the final state that has transitions only to $(n-1)$. Without loss of generality, we also assume that B is a reduced DFA.

Let $P \in Q_B$. Then the following three propositions can be easily proved:

(1) If $P \cap F \neq \emptyset$, then $0 \in P$.
(2) If $(n-1) \in P$, then $P \equiv_{L^*} P - \{n-1\}$.
(3) If $(n-2) \in P$ and $P \cap (F - \{n-2\}) \neq \emptyset$, then $P \equiv_{L^*} P - \{n-2\}$.

Using the above propositions, we can simplify the DFA B by merging all equivalent states. Let the resulting DFA be $B' = (Q'_B, \Sigma, \delta'_B, 0_B, F'_B)$. So, Q'_B has at most the following states:

(i) the starting state $0_B = \{0\}$ and the sink state $\{n-1\}$,
(ii) all P such that $P \subseteq (Q - F - \{0, n-1\})$ and $P \neq \emptyset$,
(iii) all $P = \{0\} \cup P' \cup P''$ such that $P' \subseteq (Q - F - \{0, n-1\})$ and $P'' \subseteq F - \{n-2\}$ and $P'' \neq \emptyset$,
(iv) all $P = P' \cup \{0, n-2\}$ where $P' \subseteq (Q - F - \{0, n-1\})$ and $P' \neq \emptyset$.

Note that in (iv) $P' \neq \emptyset$ because $\{0, n-2\}$ is equivalent to $\{0\}$ ($\{0\} \in F'_B$), which is included in (i).

Now we calculate the number of states in each of the items above: (i) 2, (ii) $2^{n-t-2} - 1$, (iii) $2^{n-t-2}(2^{t-1} - 1)$, and (iv) $2^{n-t-2} - 1$.

Hence we have $\#Q'_B \leq 2^{n-3} + 2^{n-t-2}$. □

As we have mentioned before, when $t = 1$, we can construct a DFA of at most $n - 1$ states to accept L^*. So, when $t = 2$ we obtain the maximum number of states for the above formula, i.e., $2^{n-3} + 2^{n-4}$.

Note that if $0 \in F$, then for each $P \in Q_B$ such that $\{0, n-2\} \subseteq P$ we have $P \equiv_{L^*} (P - \{n-2\})$. Thus, all states of (iv) are included in (iii). Then we have (i) 2, (ii) 2^{n-t-1}, and (iii) $2^{n-t-1}2^{t-2}$. The total number is $2^{n-t-1} + 2^{n-3}$. However, if $t \leq 2$, then we can construct a DFA of at most $n-1$ states to accept L^*. So, this formula reaches its maximum when $t = 3$, i.e., $2^{n-3} + 2^{n-4}$, which is the same as the one in the case $0 \notin F$.

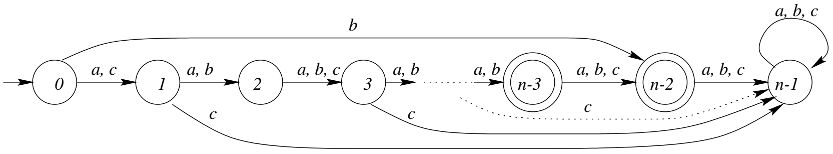
Corollary 1. *Let $A = (Q, \Sigma, \delta, 0, F)$ be a DFA accepting a finite language L, where $\#Q = n > 4$. Then there exists a DFA of at most $2^{n-3} + 2^{n-4}$ states that accepts L^*.*

Theorem 2. *There exists a DFA $A = (\Sigma, Q, \delta, 0, F)$ with $\#Q = n \geq 4$ such that any DFA recognizing $L(A)^*$ has at least $2^{n-3} + 2^{n-4}$ states.*

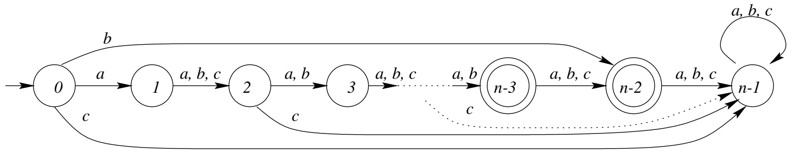
Proof. For an arbitrary integer $n \geq 4$, we define a DFA $A = (Q, \Sigma, \delta, 0, F)$, where $Q = \{0, 1, \ldots, n-1\}$, $\Sigma = \{a, b, c\}$, $F = \{n-3, n-2\}$, and δ:

$\delta(i, a) = i + 1$, for $0 \leq i \leq n-2$,
$\delta(i, b) = i + 1$, for $1 \leq i \leq n-2$, and $\delta(0, b) = n-2$,
$\delta(i, c) = i + 1$, for $0 \leq i \leq n-2$ and $n - i$ is odd,
$\delta(i, c) = n - 1$, for $0 \leq i \leq n-2$ and $n - i$ is even,
$\delta(n-1, a) = n-1$, $\delta(n-1, b) = n-1$, $\delta(n-1, c) = n-1$.

The DFA A is shown in the figure below in two cases: (a) n is odd and (b) n is even.



(a) n is odd



(b) n is even

Fig. 1. DFA A of n states such that $L(A)^*$ needs $2^{n-3} + 2^{n-4}$ states

We construct a DFA $A' = (Q', \Sigma, \delta', 0', F')$ that accepts $L(A)^*$ following the two steps described in Theorem 1: (i) construct an NFA by adding a λ-transition from each final state to the starting state; (ii) construct a DFA from the resulting NFA of the previous step using the standard subset-construction algorithm.

In the following it suffices to show that every state specified in Theorem 1 is (1) reachable from the starting state $\{0\}$ and (2) in a distinct equivalence class with respect to $L(A)^*$.

We first prove that every state in the proof of Theorem 1 is reachable. For convenience, we denote the four disjoint subsets of Q' described in (i), (ii), (iii), and (iv) of Theorem 1 by $Q'_{(i)}$, $Q'_{(ii)}$, $Q'_{(iii)}$, $Q'_{(iv)}$, respectively. In particular, we have

$$Q'_{(i)} = \{\{0\}, \{n-1\}\},$$
$$Q'_{(ii)} = \{P \mid P \subseteq \{1, \ldots n-4\} \text{ and } P \ / \emptyset\},$$
$$Q'_{(iii)} = \{P \cup \{0, n-3\} \mid P \subseteq \{1, \ldots n-4\}\},$$
$$Q'_{(iv)} = \{P \cup \{0, n-2\} \mid P \subseteq \{1, \ldots n-4\} \text{ and } P \ / \emptyset\}.$$

For $Q'_{(i)}$, obviously, the starting state $\{0\}$ and the sink state $\{n-1\}$ are both reachable. Now we prove the following claim:

Claim. Every state $q' \in Q'_{(iii)}$ is reachable (from the starting state $\{0\}$).

Let $q' \in Q'_{(iii)}$. Then $q' = P \cup \{0, n-3\}$ for some $P \subseteq \{1, \ldots, n-4\}$. We prove the claim by induction on the size of P. If $\#P = 0$, then $q' = \{0, n-3\}$. It is clear that $q' = \delta'(\{0\}, a^{n-3})$. Suppose that every state q' is reachable for $\#P = k$, $0 \leq k < n-4$. Consider the case when $\#P = k+1$. Let $q' = \{0, i_0, i_1, \ldots, i_k, n-3\}$. We know that $q'' = \{0, i_2 - i_1, \ldots, i_k - i_1, n-3-i_1, n-3\}$ is reachable by the induction hypothesis. Then it is clear that

$$\begin{aligned}
&\delta'(q'', ab^{i_1-i_0-1}a^{i_0}) \\
&= \delta'(\{0, 1, i_2 - i_1 + 1, \ldots, i_k - i_1 + 1, n-3-i_1+1, n-2\}, b^{i_1-i_0-1}a^{i_0}) \\
&= \delta'(\{0, i_1 - i_0, i_2 - i_0, \ldots, i_k - i_0, n-3-i_0, n-2\}, a^{i_0}) \\
&= \{0, i_0, i_1, i_2, \ldots, i_k, n-3\} = q'.
\end{aligned}$$

Note that if $q' = \{0, i_0, i_1, n-3\}$, let $q'' = \{0, n-3-i_1, n-3\}$. Then again $q' = \delta'(q'', ab^{i_1-i_0-1}a^{i_0})$. If $q' = \{0, i_0, n-3\}$ ($k = 0$), let $q'' = \{0, n-3\}$ and $q' = \delta'(q'', ab^{n-3-i_0-1}a^{i_0})$. Therefore, we have proved the claim.
Note that the claim directly implies that any state $P_2 \in Q'_{(iv)}$ is reachable since for any $P_2 = \{0, i_1, \ldots, i_k, n-2\}$, where $0 < i_1 < \ldots < i_k < n-3$, we have $P'_2 = \{0, i_1 - 1, \ldots, i_k - 1, n-3\} \in Q'_{(iii)}$ such that $\delta'(P'_2, b) = P_2$. Note that it is possible that $i_1 - 1 = 0$.

It is also clear that every state $P \in Q'_{(ii)}$ is reachable since for any such state $P = \{i_1, \ldots, i_k\}$, where $0 < i_1 < \ldots < i_k < n-3$, we have $P' = \{0, i_2 - i_1, \ldots, i_k - i_1, n-2\}$ such that $\delta'(P', a^{i_1}) = P$. So, we have proved that every state specified in Theorem 1 is reachable from $\{0\}$.

Now, we prove that every state above is in a distinct equivalence class of $\equiv_{L^*}$.

It is clear that if two states p and q are from different sets of $Q'_{(i)}$, $Q'_{(ii)}$, $Q'_{(iii)}$, and $Q'_{(iv)}$, then $p \ / \not\equiv$ (with respect to L^*). It suffices in the remaining to prove that if there exists $i \in \{1, \ldots, n-4\}$ such that $i \in p - q$, then $p \ / \not\equiv$ If $n-i$ is odd, then both $\delta'(p, ca^{n-i-4})$ and $\delta'(p, ca^{n-i-3})$ are final, but $\delta'(q, ca^{n-i-4})$ and $\delta'(q, ca^{n-i-3})$ cannot be final at the same time. If $n-i$ is even and $i < n-4$, then both $\delta'(p, aca^{n-i-5})$ and $\delta'(p, aca^{n-i-4})$ are final, but $\delta'(q, aca^{n-i-5})$ and $\delta'(q, aca^{n-i-5})$ cannot be both final. If $i = n-4$, then $\delta'(p, a) \in F'$ but $\delta'(q, a) \ / \in F'$. Therefore, $p \ / \not\equiv$ □

We do not yet have an example for the two-letter alphabet case. It is still open whether there exists a lower upper bound for the two-letter alphabet case.

4 Catenation of Finite Languages

We now consider the state complexity of the catenation of two finite languages.

Without loss of generality, we assume that all the DFA we are considering are reduced and ordered. A DFA $A = (Q, \Sigma, \delta, 0, F)$ with $Q = \{0, 1, \ldots, n\}$ is called an ordered DFA if, for any $p, q \in Q$, the condition $\delta(p, a) = q$ implies that $p \leq q$.

For convenience, we introduce the following notation:

$$\binom{n}{\leq i} = \sum_{j=0}^{i} \binom{n}{j}.$$

Theorem 3. *Let $A_i = (Q_i, \Sigma, \delta_i, 0, F_i)$, $i = 1, 2$, be two DFA accepting finite languages L_i, $i = 1, 2$, respectively, and $\#Q_1 = m$, $\#Q_2 = n$, $\#\Sigma = k$, and $\#F_1 = t$. There exists a DFA $A = (Q, \Sigma, \delta, s, F)$ such that $L(A) = L(A_1)L(A_2)$ and*

$$\#Q \leq \sum_{i=0}^{m-2} \min\left\{k^i, \binom{n-2}{\leq i}, \binom{n-2}{\leq t-1}\right\} + \min\left\{k^{m-1}, \binom{n-2}{\leq t}\right\}. \quad (*)$$

Proof. The DFA A is constructed in two steps. First, an NFA A' is constructed from A_1 and A_2 by adding a λ-transition from each final state in F_1 to the starting state 0 of A_2. Then, we construct a DFA A from the NFA A' by the standard subset construction. Again, we assume that A is reduced and ordered.

It is clear that we can view each $q \in Q$ as a pair (q_1, P_2), where $q_1 \in Q_1$ and $P_2 \subseteq Q_2$. The starting state of A is $s = (0, \emptyset)$ if $0 \notin F_1$ and $s = (0, \{0\})$ if $0 \in F_1$. Let us consider all states $q \in Q$ such that $q = (i, P)$ for a particular state $i \in Q_1 - \{m-1\}$ and some set $P \subseteq Q_2$. Since A_1 is ordered and acyclic, the number of such states in Q is restricted by the following three bounds: (1) k^i, (2) $\binom{n-2}{\leq i}$, and (3) $\binom{n-2}{\leq t-1}$. We explain these bounds below informally.

We have (1) as a bound since all states of the form $q = (i, P)$ are at a level $\leq i$, which have at most k^{i-1} predecessors. By saying that a state p is at level i we mean that the length of the longest path from the starting state to q is i.

We now consider (2). Notice that if $q, q' \in Q$ such that $\delta(q, a) = q'$, $q = (q_1, P_2)$ and $q' = (q'_1, P'_2)$, then $\delta_1(q_1, a) = q'_1$ and $P'_2 = \{\delta_2(p, a) \mid p \in P_2\}$ if $q'_1 \notin F_1$ and $P'_2 = \{0\} \cup \{\delta_2(p, a) \mid p \in P_2\}$ if $q'_1 \in F_1$. So, $\#P'_2 > \#P_2$ is possible only when $q'_1 \in F_1$. Therefore, for $q = (i, P)$, $\#P \leq i$ if $i \notin F_1$ and $\#P \leq i + 1$ if $i \in F_1$. In both cases, the maximum number of distinct sets P is $\binom{n-2}{\leq i}$. The number $n-2$ comes from the exclusion of the sink state $n-1$ and starting state 0 of A_2. Note that, for a fixed i, either $0 \in P$ for all $(i, P) \in Q$ or 0 is not in any set P such that $(i, P) \in Q$.

(3) is a bound since for each state $i \in Q_1 - \{m-1\}$, there are at most $t-1$ final states on the path from the starting state to i (not including i).

For the second term of $(*)$, it suffices to explain that for each $(m-1, P)$, $P \subseteq Q_2$, $\#P$ is bounded by the total number of final states in F_1. $\square$

Corollary 2. *Let $A_i = (Q_i, \Sigma, \delta_i, 0, F_i)$, $i = 1, 2$, be two DFA accepting finite languages L_i, $i = 1, 2$, respectively, and $\#Q_1 = m$, $\#Q_2 = n$, and $\#F_1 = t$, where $t > 0$ is a constant. Then there exists a DFA $A = (Q, \Sigma, \delta, s, F)$ of $O(mn^{t-1} + n^t)$ states such that $L(A) = L(A_1)L(A_2)$.*

We can simplify the formula in Theorem 3 for the case when $k = 2$, $m + 1 \geq n > 2$.

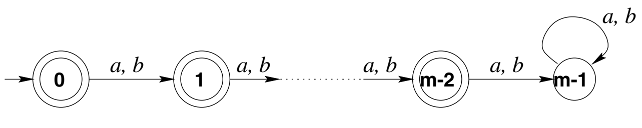
Corollary 3. *For $k = 2$ and $m+1 \geq n > 2$, the upper bound given in Theorem 3 is*

$$(m - n + 3)2^{n-2} - 1.$$

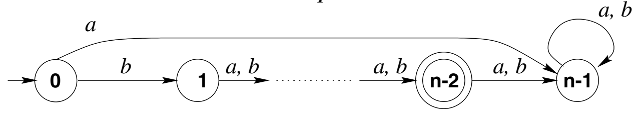
We omit the details of the mathematical calculation.

Theorem 4. *The upperbound given in Corollary 3 is reachable.*

Proof. Let $A_1 = (Q_1, \Sigma, \delta_1, 0, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, 0, F_2)$, with $\Sigma = \{a, b\}$, $Q_1 = \{0, 1, \dots, m-1\}$, $Q_2 = \{0, 1, \dots, n-1\}$, and $m + 1 \geq n > 2$. A_1 and A_2 are shown below.



A_1



A_2

Let $L = L(A_1)L(A_2)$. We show that there are at least $(m - n + 3)2^{n-2} - 1$ equivalence classes of the relation $\equiv_L$ over Σ^*.

Consider all words $w \in \Sigma^*$ such that $|w| \leq m - 2$.

If $w_1, w_2 \in \Sigma^*$, $|w_1|, |w_2| \leq m - 2$, and $|w_1| < |w_2|$, then $w_1 \not\equiv_L w_2$ since $w_1 b^{n+m-4-|w_1|} \in L(A)$ but $w_2 b^{n+m-4-|w_1|} \notin L(A)$.

Let $|w_1| = |w_2|$ but $w_1 \neq w_2$ and w_1 and w_2 differ at the ith position from the right, $i \leq n - 2$. We assume that w_1 contains an a and w_2 contains a b at that position. Then $w_1 \not\equiv_L w_2$ since $w_1 a^{n-2-i} \notin L$ but $w_2 a^{n-2-i} \in L$.

So, for each k, $0 \leq k \leq n - 2$, words of length k belong to 2^k distinct equivalence classes of $\equiv_L$. For each k, $n - 2 < k \leq m - 2$, words of length k belong to at least 2^{n-2} distinct equivalence classes.

Therefore there are at least

$$\begin{aligned} &1 + 2 + \ldots + 2^i + \ldots + 2^{n-2} + \underbrace{2^{n-2} + \ldots + 2^{n-2}}_{m-2-(n-2)+1 \text{ terms}} \\ &= 2^{n-1} - 1 + (m - n + 1)2^{n-2} \\ &= (m - n + 3)2^{n-2} - 1 \end{aligned}$$

equivalence classes of $\equiv_L$. □

5 Reversal of Finite Languages

Next we develop a tight upper bound for the state complexity of the reversal of a finite language.

Theorem 5. *Let $A = (Q, \Sigma, \delta, 0, F)$ be a DFA accepting a finite language L, where $\#Q = n \geq 3$ and $\#\Sigma = k \geq 2$. Let t be the smallest integer such that $2^{n-1-t} \leq k^t$. Then there exists a DFA $B = (Q_B, \Sigma, \delta_B, 0, F_B)$, with $\#Q_B \leq \sum_{i=0}^{t-1} k^i + 2^{n-1-t}$, that accepts L^R, i.e., the reversal of L.*

Proof. B is constructed by first reversing all the transitions of A and then determinizing the resulting NFA by the standard subset construction. Then each state in Q_B is a subset of Q. Recall that the level of a state in a finite automaton is the length of the shortest path from the starting state to this state. It is clear that the number of states at each level i of B is bounded by k^i. It is also not difficult to see that this number is bounded also by 2^{n-1-i} since they are subsets of at most $n-1-i$ states of A. Let l be the length of the longest word(s) in L (or L^R). The latter bound holds because for each i, $0 \leq i \leq l$, there exists at least one state of A that can be in a state of B of level i but not in any state of a higher level. Then the number of states at each level i is bounded by $min\{k^i, 2^{n-1-i}\}$. Since t is the smallest integer such that $2^{n-1-t} \leq k^t$, we have $\#Q_B \leq \sum_{i=0}^{t-1} k^i + 2^{n-1-t}$. Note that 2^{n-1-t} is the number of all remaining subsets of Q after the first $t-1$ levels. □

Corollary 4. *Let $|\Sigma| = 2$ and A be a DFA of $n \geq 3$ states, accepting a finite language $L \subseteq \Sigma^*$. Then there exists a DFA B that accepts L^R such that B has at most $3 \cdot 2^{p-1} - 1$ states if $n = 2p$ or $2^p - 1$ states if $n = 2p - 1$.*

Proof. Since $k = 2$, we have $2^{n-1-t} \leq 2^t$, i.e. $n - 1 \leq 2t$. If $n = 2p$ then $t = p$ and $n - 1 - t = 2p - 1 - p = p - 1$. We have

$$\sum_{i=0}^{t-1} 2^i + 2^{n-1-t} = 2^t - 1 + 2^{p-1} = 3 \cdot 2^{p-1} - 1.$$

If $n = 2p - 1$ then $t = p - 1$ and $n - 1 - t = 2p - 1 - 1 - p + 1 = p - 1$. We have

$$\sum_{i=0}^{t-1} 2^i + 2^{n-1-t} = 2^{p-1} - 1 + 2^{p-1} = 2^p - 1.$$

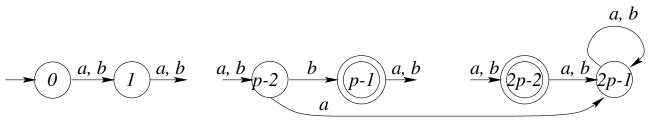
□

Theorem 6. *The bounds given by Corollary 4 are reachable.*

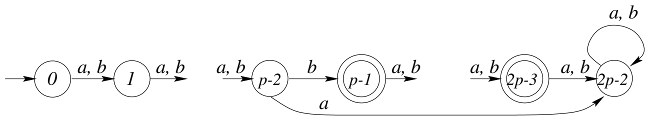
Proof. If $n = 2p$ for some integer $p > 1$, consider the DFA $A = (Q, \Sigma, \delta, 0, F)$ in the above figure.

Clearly, the reversal of A is equivalent to the catenation of A_1 and A_2 given in Theorem 4, with $m = n = p + 1$. Then any DFA accepting $L(A)^R$ has at least $2^{p-1} + 2^p - 1 = 3 \cdot 2^{p-1} - 1$ states.

If $n = 2p - 1$ for some integer $p > 1$, then look at the DFA $A' = (Q', \Sigma, \delta', 0, F')$ below:



A: n = 2p



A': n = 2p-1

The reversal of A' is equivalent to the catenation of A_1 and A_2 given in Theorem 4 with $m = p$ and $n = p + 1$. Thus, the number of states is at least $2^p - 1$. □

6 Operations on Finite Languages over a One-Letter Alphabet

We consider the case when $\#\Sigma = 1$. Without loss of generality, we assume that $\Sigma = \{a\}$.

Notice that if $A = (Q, \{a\}, 0, \delta, F)$ is a minimal DFA that accepts words of length at most l, then $\#Q = l + 1$.

Theorem 7. *Let $A_i = (Q_i, \{a\}, 0, \delta_i, F_i)$, $i = 1, 2$ be two minimal DFA, with $\#L(A_i) < \infty$, $\#Q_1 = m$, and $\#Q_2 = n$. Let $A = (Q, \{a\}, 0, \delta, F)$, $\#Q = k$, be a minimal DFA. Then we have the following:*
a) If $L(A) = L(A_1) \cup L(A_2)$, then $k = \max\{m, n\}$,
b) If $L(A) = L(A_1) \cap L(A_2)$, then $k \leq \min\{m, n\}$,
c) If $L(A) = L(A_1) - L(A_2)$, then $k \leq m$,
d) If $L(A) = L(A_1) \Delta L(A_2)$, then $k \leq \max\{m, n\}$,
e) If $L(A) = \{a\}^* - L(A_1)$, then $k = m$,
f) If $L(A) = L(A_1)L(A_2)$, then $k = m + n - 1$.
g) If $L(A) = L(A_1)^*$, then $k \leq m^2 - 7m + 13$ for $m > 4$ and $m = 3$, $k \leq 2$ otherwise.
h) If $L(A) = a \setminus L(A_1)$, then $k = m - 1$.
i) If $L(A) = (L(A_1)^R$, then $k = m$.

Proof. For a)–f) and h) the proof is obvious. For g), we give an informal proof in the following. It is clear that the length of the longest word accepted by A_1 is $m - 2$. We consider the following three cases (1) A_1 has one final state; (2) A_1 has two final states; or (3) A_1 has three or more final states. If (1), then A has $m - 1$ states. For (2), we need a lemma (Lemma 5.1 (iii)) from [9] which says that for two positive integers i and j, $(i, j) = 1$, the largest integer that cannot be presented as $ci + dj$ for any integers $c, d \geq 0$ is $i * j - (i + j)$. Let $i = m - 2$

and $j = m - 3$, i.e., $F_1 = \{m - 2, m - 3\}$. Then the length of the longest word that is not in $L(A)$ is

$$(m - 2)(m - 3) - (2m - 5) = m^2 - 7m + 11.$$

Then A has exactly $m^2 - 7m + 13$ states. If (3), it is easy to see that A cannot have more than $m^2 - 7m + 13$ states. □

Remark 1. All the above bounds are the lowest upper bounds in the worst case. If the initial DFA A_1 and A_2 are not minimal, all the above equalities become inequalities.

References

1. J.A. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events", *Mathematical Theory of Automata*, vol. 12 of MRI Symposia Series, Polytechnic Press, NY, 1962, 529-561.
2. C. Câmpeanu, "Regular languages and programming languages", *Revue Roumaine de Linguistique - CLTA*, 23 (1986), 7-10.
3. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley (1979), Reading, Mass.
4. E. Leiss, "Succinct representation of regular languages by boolean automata", *Theoretical Computer Science* 13 (1981) 323-330.
5. A. Salomaa, *Theory of Automata*, Pergamon Press (1969), Oxford.
6. K. Salomaa and S. Yu, "NFA to DFA Transformation for Finite Languages over Arbitrary Alphabets", *Journal of Automata, Languages and Combinatorics*, 2 (1997) 3, 177-186.
7. B.W. Watson, *Taxonomies and Toolkits of Regular Language Algorithms*, PhD Dissertation, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1995.
8. D. Wood, *Theory of Computation*, John Wiley and Sons, 1987.
9. S. Yu, Q. Zhuang, K. Salomaa, "The state complexities of some basic operations on regular languages", *Theoretical Computer Science* 125 (1994) 315-328.
10. S. Yu, Q. Zhuang, "On the State Complexity of Intersection of Regular Languages", *ACM SIGACT News*, vol. 22, no. 3, (1991) 52-54.
11. S. Yu, Regular Languages, in: *Handbook of Formal Languages, Vol. I,* G. Rozenberg and A. Salomaa eds., Springer Verlag, 1997, pp. 41–110.

Treatment of Unknown Words

Jan Daciuk

Department of Applied Informatics, Technical University of Gdańsk,
Ul. Narutowicza 11/12, PL80-952 Gdańsk, Poland
jandac@pg.gda.pl

Abstract. Words not present in the dictionary are almost always found in unrestricted texts. However, there is a need to obtain their likely base forms (in lemmatization), or morphological categories (in tagging), or both. Some of them find their ways into dictionaries, and it would be nice to predict what their entries should look like. Humans can perform those tasks using endings of words (sometimes prefixes and infixes as well), and so can do computers. Previous approaches used manually constructed lists of endings and associated information. Brill proposed transformation-based learning from corpora, and Mikheev used Brill's approach on data for a morphological lexicon. However, both Brill's algorithm, and Mikheev's algorithm that is derived from Brill's one, lack speed, both in the rule acquisition phase, and in the rule application phase. Their algorithms handle only the case of tagging, although an extension to other tasks seems possible. We propose a very fast finite-state method that handles all of the tasks described above, and that achieves similar quality of guessing.

1 Introduction

A morphological analysis of words in a text is needed in many applications. It constitutes a prerequisite for natural language parsing and all applications that use it, and it is also useful in document retrieval. Such analysis is usually lexicon-based, i.e. it requires a morphological lexicon.

Unfortunately, real-world texts contain correct words that cannot be found in a lexicon. It seems impossible to record all words of a living language in a lexicon, as a lexicon is static in nature, and a language is a living thing – new words are coined continually. Another reason for finding words not present in the lexicon is the Zipf's law[1]. The Zipf's law states that the rank of an element divided by the frequency of occurrence is constant. E.g. in the Brown corpus, two percent of different words[1] account for sixty nine percent of the text. About seventy five percent of different words occur five or fewer times in the corpus. Fifty eight percent of different words occur two or fewer times, and forty four percent only occur once. The consequence of the Zipf's law is that by doubling the number of words in the lexicon, one gets only a few percents of the coverage

[1] Eric Brill uses the terms *word type* and *token*

of an arbitrary unrestricted text. Therefore, increasing the size of the lexicon is a very costly effort yielding minute results.

New words are also constructed by derivation or compounding. The number of potential words formed in that way is huge. Therefore, it is not practical to store all such derivatives and compounds in the lexicon. In many cases there may be many ways to form a new word, and it is not possible to predict which one would be chosen by humans as correct.

Additionally, texts may contain incorrect words. For purpose of e.g. spelling corrections, the morphological categories of a misspelled word may help reduce the list of possible corrections. If the misspelling does not affect the word's flectional ending (and its prefix, if present), these categories may still be easily obtainable from the corrupted version.

Some of previously unseen words eventually make their way into the (morphological) lexicon. In order to do that, we need to give them morphological descriptions. The task of writing them by hand is laborious; it would be much easier to choose a description from a list of feasible descriptions.

Humans can perform all those tasks quite well. The reason they can do that is that they can associate information they want to extract with endings of words (sometimes also with their prefixes and infixes). So can do computers. We propose a fast finite-state technique to accomplish that.

2 Related Work

In the past, various hand-crafted heuristics have been used for the purpose of morphological analysis of unknown words. Later, they have been supplemented by statistical techniques (e.g. [7]). However, although the probabilities of different endings leading to their corresponding categories were calculated, the endings themselves were chosen manually. A revolutionary approach was proposed by Eric Brill ([1], [2]). The endings, as well as prefixes, are found by the program. Unknown words are first tagged by a naive initial-state annotator that assigns the tags proper noun or common noun on the bases of their capitalization. Then five types of transformations are applied:

Change the tag of an unknown word (from X) to Y if:

1. Deleting the prefix (suffix) x, $|x| \leq 4$, results in a word (x is any string of length 1 to 4).
2. The first (last) (1,2,3,4) characters of the word are are x.
3. Adding the character string x as a prefix (suffix) results in a word ($|x| \leq 4$).
4. Word w ever appears immediately to the left (right) of the word.
5. character z appears in the word.

The result is compared with a tagged corpus. The best-scoring rule is chosen, and applied to the corpus so that it becomes new input data. The learning stops when no rule can increase the score. The transformation type 4 takes into account the context of the unknown word. When the morphological analysis is separated from a contextual tagger, as it is the case with our approach, the tagger must find those rules itself. The transformation types 1 and 3 checks whether adding or deleting characters from/to a word results in another word. In our algorithm those transformations are not present, as well as transformation type 5, which can be treated, if necessary, as a supplementary heuristics. The rules schemata presented above do not account for infixes, which can be treated with our method.

Andrei Mikheev ([3]) applied Brill's transformations to the data for a (pre-existing) morphological lexicon. He uses the following template:

$$G =_{x:\{b,e\}} [-S \;\; + M \;\; ?I\text{-class} \rightarrow R\text{-class}],$$

where:

- x indicates whether the rule is applied to the beginning or end of a word and has two possible values: b – beginning, e – end;
- S is the affix to be segmented; it is deleted (-) from the beginning or end of an unknown word according to the value of x;
- M is the mutative segment (possibly empty), which should be added (+) to the result string after segmentation;
- i-class is the required POS-class (set of one or more POS-tags) of the stem; the result string after the -S and $+M$ operations should be checked (?) in the lexicon for having this particular POS-class; if I-class is set to be "void" no checking is required;
- R-class is the POS-class to assign ($\rightarrow$) to the unknown word if all the above operations (-S $+R$?I) have been successful

Compared to Brill's algorithm, Mikheev checks also (optionally) the morphological class (a set of categories) of the resulting word in transformation templates 1 and 3. Also, his algorithm returns all categories of an unknown word, not only the most probable one.

Although Mikheev is not aware of that fact[2], the rules learnt by Brill's algorithm can be transformed into a finite-state automaton. However, that process is complex and time-consuming. The learning process takes time as well. Our algorithm produces the FSA directly from data exploring the links that occur naturally in the format of data we use. In Brill's approach, copied by Mikheev, the length of suffixes and prefixes is a constant. Increasing it means much more computation. In our method, the suffixes are discovered naturally, so there is no need to limit their lengths.

It should be stressed that both Brill and Mikheev guess only categories (tags), and not base forms or morphological descriptions. Jan Tokarski (see [6], in Polish) prepared data for guessing not only categories, but the base forms and morphological descriptions as well. However, he did that enormous work by hand. The result is a book of a few hundred pages. It has been used in creation of several morphological dictionaries of Polish. In at least one implementation, the contents of the book has been converted into a finite-state automaton.

[2] [5] does not appear among references in [3]

3 Finite-State Approach

Our approach is based on the observation that it is possible to associate endings with required information by inverting inflected words, and sticking the required information (an *annotation*) at the end. By performing that operation on all inflected forms in the lexicon, we get a finite set of strings. Therefore, it is possible to convert it into a minimal, deterministic, acyclic, finite-state automaton that we call a *guessing automaton*. To find appropriate annotation for unknown word, we need to invert the word, and then look for it in the guessing automaton.

3.1 Data

The exact format of the annotation depends on the information we want to put into it. In a general case, we need a special symbol that we call an *annotation separator* to separate annotations from the inflected forms. For reasons that we explain later, we also need another special symbol to mark the end of the inverted inflected form; we put that symbol in front of the annotation separator. If the annotation should consist of morphological categories of the inflected form, we simply put them after the annotation separator. Example:

```
abmob_+Verb[mode=ind tense=past num=sg person=3]
```

If the annotation is to be the base form, a little bit of coding is necessary in order to avoid inflating the automaton. We assume that it is only the ending that is different in the base form as compared with the inflected form. Therefore, we can replace the full base form with a code that says how many characters are to be deleted from the end of the inflected form, and a string consisting of characters that are to be appended to obtain the base form. When no characters are to be deleted, we put 'A' there, one character – 'B', etc. Example:

```
abmob_+Ber
```

It is possible to put both the base form, and the categories in annotations. Example:

```
abmob_+Ber+Verb[mode=ind tense=past num=sg person=3]
```

Annotations for morphological data acquisition are more complicated, as the base form may be different form the lexical form, and the lexical form may contain arch-phonemes. Also, they depend on the particular morphology program we use. Therefore, we will not give any examples of that here. The annotation formats we present here are very simple; however, by modifying the annotation format we can successfully handle prefixes and infixes as well.

3.2 Pruning

As word endings normally decide what annotation should be associated with a given word, the automaton has a particular structure. For a given word, the first few states have many outgoing transitions. Then, there is a chain of states linked with each other with single transitions. Passing to annotations, a more complicated transition network appears again. As for any state in the central part, there is only one way leading from the state to the appropriate annotations, it represents no useful information for our purpose. So all states from that part can be pruned, along with the corresponding transitions. Pruning explains the need for the special symbol marking the end of the inverted inflected form (the beginning of the inflected form). We need it because we no longer have full words in the automaton, sometimes entire shorter words may constitute the end part of longer words, and different words may have different annotations, so we need to distinguish them.

Pruning is governed by the following rules:

Rule R 1. The pruning process does not apply to transitions belonging to annotations.

This rule should be obvious, because annotations are what we want to obtain in the recognition phase. The transitions that are pruned belong to the inflected forms, and more precisely: to their beginnings that do not influence the annotations.

Rule R 2. A transition can be removed only if the pruning process has already visited all transitions that can be reached through the target state, except the transitions representing annotations. In other words, the states are visited (and the outgoing transitions pruned if possible) in the postorder method.

This means that we traverse the automaton recursively in depth, cutting unneeded transitions on the way back.

Rule R 3. A transition cannot be removed if the target state has a transition that does not belong to annotations, but cannot be removed.

This means that the target state has transitions that distinguish between different annotations, i.e. they lead to different sets of annotations. We do not want to lose that distinction.

Rule R 4. A state can always be replaced with an equivalent one (i.e. a state with the same transitions leading to the same states).

The automaton should be kept minimal.

Rule R 5. A state with all transitions leading to one state (the target state) can be removed with all transitions, and transitions that point to it should be replaced by transitions pointing to the target state.

This is the rule that actually cuts transitions. Note that it also applies to states with one only transition.

The consequence of the rules R2 and R5 is that we cannot remove states (and transitions that lead to them) if they have transitions leading to different sets of annotations. This is because the rule R2 ensures that all transitions of a visited state lead to states that either cannot be removed, or that have an outgoing transition labeled with the annotation separator.

The process described above leads to the construction of a finite state automaton that contains all required information. However, the automaton is still big, and an effort can be made to further reduce its size. Looking at its contents, we can see many states with a majority of transitions leading to one state (we will use the term *default state*), but with other transitions as well. The target state must have one outgoing transition, and it must be labeled with the annotation separator. We can treat less frequent transitions as exceptions, assuming that *all* other transitions, even those that had not appeared in our lexicon, lead to the default state. Acting under this assumption, we can replace the frequent transitions and the default state with the transition leading from the default state. A limit can be imposed on the ratio of frequent/less frequent transitions to trigger the pruning.

There is a difference between rules R1, R2, . . . R5, and the rule we are about to introduce (R6). Rule R6 introduces a generalization. While still 100% of words present in the lexicon are annotated correctly, R6 may select one annotation among many as the correct one, and hide other possibilities. This may speed up an annotation process, but it can also introduce errors: some correct (but less probable) possibilities may not be shown, as the lexicon may not contain data that associates their endings with their correct annotations.

Rule R 6. If for a given state the number of outgoing transitions leading to one state (the default state) is greater or equal to the number of all other outgoing transitions multiplied by a small integer, and the default state has only one outgoing transition and it is labeled with annotation separator, then the default state can be removed, and all transitions that lead to it should be replaced by the transition leaving the default state.

Sometimes, it is impossible to devise a rule that associates an ending with the correct annotation, because the choice is lexicalized, i.e. it depends on a particular word, and it seems arbitrary from the morphological point of view. For example, in Polish, there is a rule that transforms adjectival endings *-sny* in base forms into *-śniejszy* in comparatives and superlatives. There is, however, another rule that transforms endings *-śny* into *-śniejszy* in comparatives and superlatives. So there is no other way of knowing what the base form might be from a comparative or superlative ending other than a dictionary lookup. R6 introduces artificial divisions, e.g.:

-raśniejszy → -raśny	jaśniejszy → jasny
-maśniejszy → -maśny	-iaśniejszy → -iasny
-waśniejszy → -waśny	-dośniejszy → -dosny
-ośniejszy → -ośny	-ześniejszy → -zesny
-bleśniejszy → -bleśny	-oleśniejszy → -olesny
-uśniejszy → -uśny	

while the right answer is that both annotations must be considered:

-śniejszy → -śny
-śniejszy → -sny

To cope with that situation, we introduce a new rule that strives to accommodate such cases. We will use the term *first annotated state* to name a state that is a target of a transition labeled with the annotation separator (a state that begins an annotation or a set of annotations).

Rule R 7. If for a given state the number of first annotated states that are reachable from the given state does not exceed a given limit, then:

– replace the first annotated states by their union;
– replace all the states and transitions between the chosen state and the union of the first annotated states by a single transition labeled with the annotation separator.

Note that it is possible to introduce a lower limit on the number of states to be removed in order to insure that we are dealing with a case such that the one described above (*sny* and *śny*). The rule can then work in parallel with R6.

It is worth noting that while the rule R6 introduces very detailed distinctions, R7 discards details. For the guesser, the result of applying R7 is that one gets more choices than without having applied R6 or R7. As to the lexicon size, R7 removes small differences between similar word forms, making it possible to infer more general and compact relations between endings and annotations.

Please note that although no annotation possibility is lost, and the automaton is much smaller, the answers for known words are no longer 100% accurate. The correct answer appears always, but it may be accompanied by other, incorrect possibilities. In many cases exceptions are merged with regular rules. A lower limit imposed on the number of states to be removed by this rule can solve the problem.

3.3 Recognition

It is mostly endings that decide what annotations a word may have. To get an annotation, we invert the unknown word, put the special symbol (word beginning marker) at the end, and look for such string in the automaton. Sooner or later, we come to a state that has no transition labeled with the subsequent letter of the string. That state may have a transition labeled with the annotation separator. If it has one, then the right language of that state is the set of annotations for the word. If not, we recursively look at the descendants of the state, searching for states with transitions labeled with the annotation separator, and adding the right languages of the states they lead to to the resulting annotation of the unknown word.

When we need to code prefixes or infixes (e.g. for German), the annotations contain information on what the prefix should be, and on what should be done at the beginning of the word to obtain the base form. In that case, the prefix stored in the automaton must be compared with the prefix of the analysed word.

Table 1. Recall and precision for predictions of properties of unknown words

	categories only			categories and base forms		
	R1-R5	R1-R6	R1-R5,R7	R1-R5	R1-R6	R1-R5,R7
recall	94.57	93.92	97.80	93.43	92.74	95.93
precision	90.03	91.64	64.81	88.45	90.34	61.42

4 Results

Experiments were carried out on French morphology from ISSCO, Geneva, Switzerland. The data for the morphological lexicon was divided in 10 parts, 9/10 were used to construct guessing automata, and 1/10 as a source of words whose annotations were to be guessed. This was done ten times, i.e. for each pair (9/10, 1/10). Standard measures of recall, precision, and coverage were used to evaluate the results. Recall and precision were calculated for each item, and the average of all items was calculated for each part.

Table 1 shows recall and precision for prediction of morphological categories, and both morphological categories and base forms. The coverage is 100% in all cases. For the rule R6, we chose that there should be twice as many transitions leading to the default state than to other states. For the rule R7, we merged only two states at a time, they had to be the only children of a given state, and we did not count the transitions that led to them. By setting a threshold on the minimum number of transitions leading to the states that are to be merged by the rule R7, we can raise precision, but lower the coverage (e.g. to 95.99% and 75.39% for categories only, and to 94.48% and 74.10% for both categories and base forms).

The results show that if we want to minimize the number of choices and raise precision, we should use rules R1-R6. This is the case of simple POS-tagging. In cases where we want to make sure that we do not miss any possibility, we should use rules R1-R5 and R7.

Mikheev ([3]) claims achieving 95.24% recall, 85.16% precision, and 92.66% coverage on categories only (he did not consider base forms). Note that we have 100% coverage, so his recall and precision should probably be multiplied by 0.9266 in order to be compared directly to ours. However, he performed experiments on English words, and it is not clear what the impact of the chosen language is on the results. He also used smoothing on a corpus. Since we have access neither to his programs, nor to the data he used, the only way we could compare our results to his would be to emulate his approach on our data. Our experiments required ca. 14 minutes to build all 10 guessing automata for one set of rules, and 2 minutes 15 seconds to evaluate the results on Pentium II 350 MHz with 128 MB memory running under Linux. Using Mikheev's method would probably take days.

Table 2 shows the results for guessing morphological descriptions of words. The coverage is no longer 100%; this is probably caused by existence of arch-

Table 2. Recall, precision, and coverage for predictions of morphological descriptions of unknown words

	R1-R5	R1-R6	R1-R5,R7
recall	93.22	92.47	94.67
precision	86.26	88.27	66.31
coverage	99.94	99.96	99.99

Table 3. Size of the automaton as function of rules

	categories only			morphological descriptions		
	R1-R5	R1-R6	R1-R5,R7	R1-R5	R1-R6	R1-R5,R7
states	19800	18507	10123	39832	38301	27608
transitions	65635	48419	32729	99535	78976	61077

phonemes. The format of descriptions was the one used by mmorph tool ([4]) from ISSCO, Geneva.

The choice of rules influences the size of the automaton. Table 3 shows that relation for guessing only categories, and for guessing morphological descriptions. We can see that R6 cuts mostly transitions, but R7 cuts both transitions and states.

5 Conclusions

We have presented a novel finite-state approach to morphological analysis on words that are not present in the lexicon. Its main adavantage is speed, both in the rule acquisition phase, and in the rule application phase. Our method can also be used for lemmatization, and for acquisition of new words for a morphological lexicon.

All programs and data used in our experiments are publically available. The finite-state guesser is available form `http://www.pg.gda.pl/~jandac/fsa.html` as part of the *fsa* package, while mmorph and MULTEXT French morphology (written by Pierrette Bouillon) is available from ISSCO, Geneva, Switzerland, at `http://www.issco.unige.ch/`.

References

[1] Eric Brill. *A Corpus-Based Approach to Language Learning.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, USA, 1993.

[2] Eric Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565, December 1995.

[3] Andrei Mikheev. Automatic rule induction for unknown-word guessing. *Computational Linguistics*, 23(3):405–423, September 1997.

[4] Dominique Petitpierre and Graham Russell. MMORPH - the Multext morphology program. Technical report, ISSCO, 54 route des Acacias, CH-1227 Carouge, Switzerland, October 1995.

[5] Emmanuel Roche and Yves Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–253, June 1995.

[6] Jan Tokarski. *Schematyczny indeks a tergo polskich form wyrazowych.* Wydawnictwo Naukowe PWN, 1993.

[7] Ralph Weischedel, Marie Meteer, Richard Schwartz, Lance Ramshaw, and Jeff Palmucci. Coping with ambiguity and unknown words through probabilistic models. *Computational Linguistics*, 19(2):359–382, 1993.

Computing Entropy Maps of Finite-Automaton-Encoded Binary Images

Mark G. Eramian

Department of Computer Science
University of Western Ontario
London, Ontario, N6A 5B7, Canada
meramian@csd.uwo.ca

Abstract. Finite automata are being used to encode images. Applications of this technique include image compression, and extraction of self similarity information and Hausdorff dimension of the encoded image. Jürgensen and Staiger [7] proposed a method by which the local Hausdorff dimension of the encoded image could be effectively computed. This paper describes the first implementation of this procedure and presents some experimental results showing local entropy maps computed from images represented by finite automata.

1 Introduction

Local entropy (Hausdorff dimension) measures of images are of interest because the local entropy of an image is closely related to local texture. If we can map texture, it is possible to map the boundaries of objects in the image, or to simply map relative texture differences over the image. Image texture mapping techniques have many applications in remote sensing, an example of which can be found in [5]. A general survey of texture analysis techniques for various applications is given in [13] and [8].

A method of computing the global Hausdorff dimension of a two color (binary) image from the finite automaton representation was proposed in [11]. This measure is not appropriate for inferring local texture information, since such a measure corresponds only to the most disorderly point in the image. This idea was refined, and a measure of local entropy, as well as an effective method of computing local entropy, was defined in [7]. This paper introduces the first implementation of the afforementioned method and the first image entropy maps computed using this method.

In the sequel, we first briefly explain the relationship between languages, automata, and images. We then proceed to review some of the theory behind the localization of Hausdorff dimension, the proposed procedure for computation of the local entropies from the automaton, and then discuss the actual implementation and software, and show some example entropy maps.

2 Languages, Automata, and Images

Consider a binary image given by a finite quadtree. Suppose each edge of the tree is labeled with a number from the set $\Sigma = \{0, 1, 2, 3\}$ representing the corresponding quadrant. The labels along a path from the root of the tree to a leaf node form a word over the alphabet Σ. Each such word can be thought of as the quadtree address of a pixel that is turned "on". All such words together form a finite regular language and we can thus construct the corresponding automaton. If a word is accepted by the automaton, then the pixel at the corresponding address is turned "on".

Suppose an image is given as an infinite quadtree. The quadtree addresses of image points that are turned "on" form a ω-language $M \subseteq \Sigma^\omega$. If the language M is regular we can then construct a Büchi automaton [12] $A = (Q, q_0, \Delta, F)$ which accepts M. If we then treat A as a classical automaton with $F = Q$ and run all finite words of length n on A we can render the original image at a finite resolution of $2^n \times 2^n$ pixels.

A detailed and more generalized description of how to encode greyscale images as finite automata is given in [2] and [1].

3 Local Hausdorff Dimension

In this section we review how the local Hausdorff dimension measure is obtained from the global Hausdorff dimension, as described in [7].

Let M be a subset of $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Let $\text{pref}_n M$ be the set of prefixes of length n of M. The language

$$w^{[-1]}M = \{\xi \mid \xi \in \Sigma^\infty, w\xi \in M\} \tag{1}$$

is called a state of M. M is finite-state if it has finitely many states. The entropy of M is denoted by H_M (which can be computed from the structure function) and the Hausdorff dimension of M is denote $\dim M$. It is well known that $\dim M \leq H_M$ for every $M \subseteq \Sigma^\omega$. In fact, for finite-state and closed ω-languages, it is true that $\dim M = H_M$. The following theorem shows how to calculate the global entropy, and hence the global Hausdorff dimension of M.

Theorem 1. [7] *If M is a finite-state closed ω-language, then $H_M = \log_{|\Sigma|} \alpha$ where α is the maximum eigenvalue of the adjacency matrix U_M of M.*

Since $\dim M$ is constructed from a metric space, it's localization begins with defining local size measures. Let $S = (X, \varrho)$ be any compact metric space, and let λ be a real valued function on S which satisfies:

$$\lambda(M) \geq 0 \text{ for } M \not= \emptyset \tag{2}$$

$$\lambda(X) < \infty \tag{3}$$

$$\lambda \text{ is monotone} \tag{4}$$

A function λ with these properties is called a size measure. λ could be any of entropy (H), subword complexity (τ), program complexity (κ) or Hausdorff dimension (dim). These measures are localized by applying the localization operator $l^{(\lambda)}$. For $M \subseteq X$, $x \in X$, and $\epsilon > 0$, let

$$l_M^{(\lambda)}(x, \epsilon) = \lambda(K_\epsilon(x) \cap M) \tag{5}$$

where $K_\epsilon(x)$ is the open ball of radius ϵ around point x, or, all points x' such that $\varrho(x, x') < \epsilon$.

Note that if $M \subseteq M'$ and $K_\epsilon(x) \subseteq K_{\epsilon'}(x')$ then $l_M^{(\lambda)}(x, \epsilon) \leq l_{M'}^{(\lambda)}(x', \epsilon')$. This property causes the following definition to make sense:

Definition 1. *For $M \subseteq X$ and $x \in X$,*

$$l_M^{(\lambda)}(x) = \lim_{\epsilon \to 0} l_M^{(\lambda)}(x, \epsilon) \tag{6}$$

is the local (M, λ)-size-measure at x.

It turns out that, as in the non-local case, several important local measures coincide if M is finite-state and closed.

Theorem 2. [7] *Let $M \subseteq \Sigma^\omega$ be finite-state and closed. Then*

$$l_M^{(\text{dim})}(\xi) = l_M^{(H)}(\xi) = l_M^{(\tau)}(\xi) = l_M^{(\kappa)}(\xi) \tag{7}$$

for all $\xi \in \Sigma^\omega$.

The fact that local entropy and local Hausdorff dimension coincde for such languages makes it possible to compute the Hausdorff dimension of the states of M by computing the entropy of the states of M.

Theorem 3. [7] *Let M be a finite-state closed ω-language and let ξ be an ω-word. If $(\text{pref}_i\xi)^{[-1]}M$ is empty for some $i \in \mathbb{N}$ then $l_M^{(H)}(\xi) = 0$. Otherwise, there is a state M_ξ occuring infinitely often in the sequence $\left((\text{pref}_i\xi)^{[-1]}M\right)_{i\in\mathbb{N}}$ and $l_M^{(H)}(\xi) = H_{M_\xi}$.*

Computing the local entropy at ξ, and accordingly at the point of the image addressed by quadtree address ξ if $M \subseteq \{0, 1, 2, 3\}^\omega$, can now be performed by two steps:

1. Find a state M_ξ which occurs infinitely often among the states $\left((\text{pref}_i\xi)^{[-1]}M\right)_{i\in\mathbb{N}}$.
2. Compute H_{M_ξ}.

If ξ is of the form uv^ω for some $u, v \in \Sigma^*$ then step 1 can be performed in finite time. The next section discusses the proposed method for performing step 2.

4 State Entropy Computation Procedure

Given the results of the previous section, the authors of [7] suggest the following method for computing H_{M_ξ}.

Consider the directed multigraph G_M. This graph corresponds to the automaton $A^M = (Q, \Sigma, \delta, s, F)$ of the language M as follows. The edges are labeled with elements of Σ and the set of vertices S_M are the non-empty states of M. There is an edge (Q, x, Q') from Q to Q' with label x if and only if $Q' = x^{[-1]}Q$, that is, if there is a transition from state Q to state Q' on input x. The set of edges we call E_M. We write $Q \vdash Q'$ if there is an edge from Q to Q', and we let $\vdash^*$ be the reflexive and transitive closure of $\vdash$. A non-empty subset K of S_M is a strongly connected component of G_M if for any two states $Q, Q' \in K$, one has $Q \vdash^* Q' \vdash^* Q$. We extend $\vdash$ to components K, K' by writing $K \vdash K'$ if and only if $Q \vdash Q'$ for some $Q \in K$ and $Q' \in K'$.

The states are then numbered according to the following rules:

1. M_s, the start state, is state number 1.
2. The states in a strongly connected component are numbered consecutively.
3. If Q and Q' are in different strongly connected components and $Q \vdash^* Q'$ then the number of Q is strictly less than that of Q'.

This numbering induces a numbering of the strongly connected components. Under the new numbering of states the adjacency matrix U_M of M will be in upper block diagonal form:

$$U_M = \begin{pmatrix} A_1 & \cdots & \cdots \\ 0 & \ddots & \cdots \\ 0 & 0 & A_k \end{pmatrix} \tag{8}$$

where k is the number of strongly connected components and A_i corresponds to the transitions within component i. H_{M_ξ} is obtainable using the following theorem.

Theorem 4. *If M is finite-state and closed and K is a strongly connected component of G_M and $Q \in K$ then*

$$H_Q = \log \max\{\alpha_{K'} \mid K \vdash^* K'\} \tag{9}$$

where $\alpha_{K'}$ is the maximal eigenvalue of $A_{K'}$.

The computation of the entropy of the states of M is reduced to the computation of the eigenvalues of each A_i of U_M.

5 State Entropy Computation Implementation

For the first time, the proposals reviewed in the previous section have been implemented, and we describe this implementation here. A suite of software has been written by the author in C. There is program for precisely encoding a binary image as an automaton, a program to compute the state entropies of that automaton, and a third program to render both the automaton-generated image and the entropy map image at any resolution that is a power of two.

The program that encodes images as an automaton is very similar to the algorithm given in [3] for greyscale images, except it does not consider the simple transformations on the subimages.

The state entropy computation program is based on the theory presented in the previous sections and is described in more detail here. The rendering program will be discussed in the next section.

In this section, we restrict M to be a language of image quadtree addresses over the alphabet $\Sigma = \{0, 1, 2, 3\}$. From a finite quadtree, we infer the graph G_M. G_M corresponds precisely to the classical finite automaton $A^M = (Q, \Sigma, \delta, s, F)$ that encodes the image. Inference algorithms have already been published for grayscale images [2] and are easily adapted to black and white images.

The state entropy calculation program reads in the infered automaton and constructs G_M. The strongly connected components (or simply components) are computed and each state is tagged with its resulting component number.

A new graph $T_M = (S_T, E_T)$ is then constructed by representing all of the states in a component by a single vertex. S_T thus contains an element for each component K in G_M and E_T contains an edge (K, K') if and only if $K \vdash K'$ in G_M.

It is easy to show that T_M is a tree. Since T_M is a tree, we can perform a topological sort on the tree. As we order each vertex of T_M, we assign consecutive *topological numbers* to the states in the corresponding component of G_M. At the conclusion of this numbering, the topological numbers are consecutive both within each component and over all of the states in G_M. During this step we also note the smallest topological number t_i in each component, and construct a vector L such that t_i is stored in L_i.

From the automaton, it is easy to build the adjacency matrix U_M, one simply counts the number of transitions between each ordered pair of states. Instead of the original state numbers, however, we use the topological numbers of the state to index the adjacency matrix. This results in the upper block diagonal form matrix shown in equation 8.

Using the component size vector L we can now easily extract the submatrices A_i of U_M. Specifically,

$$A_i = \begin{pmatrix} (U_M)_{aa} & \cdots & \cdots \\ \vdots & \ddots & \ldots \\ \vdots & \cdots & (U_M)_{bb} \end{pmatrix} \tag{10}$$

where $a = L_i$, and $b = L_{i+1} - L_i$.

The next step is to compute for each A_i the maximal eigenvalue α_i, if it exists. Since these matrices are non-negative and irreducible, the Perron-Frobenius theorem (see [10]) applies, which states that each such matrix has a unique positive and real eigenvalue. Our interest is in the maximal eigenvalue α_i of A_i, so the power method of computing eigenvalues, as described in [6], lends itself nicely to the problem, since, when the power method converges, it always converges to the largest eigenvalue. The power method will converge for our matrices because they are non-negative.

Having computed each α_i, it is now possible to compute the state entropies of M according to Theorem 4. Essentially this theorem requires that, to find the entropy H_q of state $q \in Q$, we find the component K' with the largest submatrix eigenvalue that is reachable from state q, that is, $\alpha_{K'}$, and take the logarithm of that eigenvalue. This can be simplified further, since, in component K, every state in K is reachable from every other state in K. Therefore, if we find H_p for any one state p in K, then we have found H_q for all $q \in K$. This reduces this step to finding for each *component* K, the component K' with the largest matrix eigenvalue that is reachable from K (which could be K itself), then assigning $\alpha_{K'}$ to each state $q \in K$. K' can be found easily from T_M using a depth first search algorithm. The resulting state entropies for all $q \in Q$ are output to a data file for later consumption by the entropy map renderer.

6 Entropy Maps

As observed in Sect. 2, if we allow all of the states in G_M to be final states, then we can render the image at any 2^n by 2^n reolution by determining the acceptance or non-acceptance of all finite words in Σ^n. The rendering program thus reads in the automaton G_M, and the state entropies are read into a vector H. If a word $w = w_1 w_2 \dots w_n$ is accepted, then the pixel at the quadtree address w is turned "on". The entropy map is constructed by doing exactly the same thing except instead of turning "on" the pixel of accepted quadtree address w, we mark the corresponding pixel with the value of H_q where q is the state that accepted w.

If w is rejected at the input w_i (due to an absent transition), then the address w is assigned an entropy value of

$$\left(\frac{1}{2^{n-i}}\right) H_q \tag{11}$$

where q is the current state when w_i is read. The same is true if w_i results in a transition to the state which represents the image with all pixels turned "on". This is done because in both cases, the automaton is about to make a transition to a state that represents either the all black image, or the all white image. We call these "flat" states. Both such states have an entropy of zero. This does not make sense when you consider, for example, a black and white pixel checkerboard pattern which intuitively should not have an entropy of zero. This problem arises because we are using an automaton generated from a finite quadtree. The inverse exponential coefficient compensates for this. The larger i is, the more closely the coefficient approaches unity. Thus if we reach a flat state at the pixel level, where $i = n$, then the pixel in question will receieve the full non-zero entropy value of the previous state. If we reach a flat state early in the reading of w where $i \ll n$ (which corresponds to a large flat area in the image) then we get an entropy value that is close to zero because the coefficient is very small. This is exactly what we would expect intuitively.

The entropies in H are, in general, not integers; hence we we cannot directly generate an image from them. We can, however, before assigning entropies to pixels, map the range of entropy values in H into the range $[0, 255]$. This will produce a greyscale image where greylevel is an indication of relative entropy. We apply the following transformation $g : H \to [0, 255]$ to each $h \in H$:

$$g(h) = \left\lfloor 255 \times \left(\frac{h - h_{min}}{h_{max} - h_{min}} \right) \right\rfloor \tag{12}$$

where h_{min} and h_{max} are the largest and smallest elements in H respectively.

Having assigned entropy values to each quadtree address and applied g to each such value we then create an image by converting the quadtree addresses to pixel coordinates, and assign the transformed entropy values to the pixels. The result is an greyscale image which shows how local entropy varies over the original image. For further enhancement, one can use the 8-bit greylevels to index a color palette and the image can be converted into a false color entropy map.

It should be noted that if one desires to compare entropy maps of two different images visually, then g is not an appropriate mapping because it normalizes the raw entropy values. The same greylevel in two different entropy maps will not necessarily represent the same actual entropy value.

We conclude this section with a few example entropy maps.

Example 1. Figure 1 shows a binary image of the author which was generated by a finite automaton, and the corresponding local entropy map. White indicates the areas of highest entropy, while black represents the areas of lowest entropy. A notable feature of this entropy map are that flat black and flat white regions, such as the image background and teeth area respectively, are both represented by areas of zero entropy. Also of interest is that the lights in the background, which are approximately flat white areas on a flat black background, show up as having high entropy only along the edge of the light, white the flat areas on either side of the edge show very low entropy. Finally one notes that the gradient from the shit collar to the shoulder also shows up as a slight gradiant in entropy.

Example 2. In figure 2 we see an example of a hand-designed automaton to illustrate that the strong components really do result in regions of different entropy. The automaton that generates the picture on the left has four different components, each generating one quadrant of the image. Although difficult to see without color enhancement, each quadrant shows a different entropy value in the right hand image. Addtional entropy levels are introduced by the exponential coefficient discussed above.

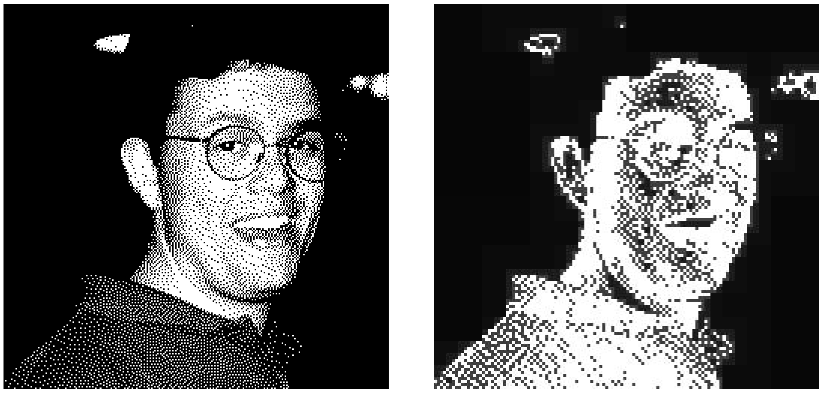


Fig. 1. *Left:* a picture of the author as generated by a finite automaton. *Right:* the local entropy map of the left image.

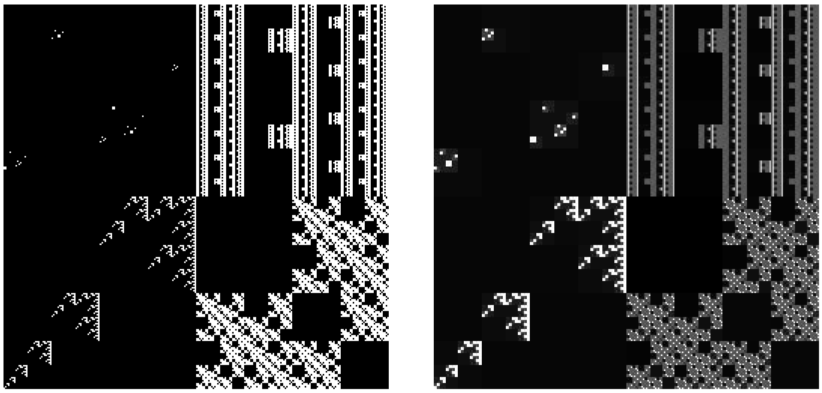


Fig. 2. *Left:* A hand-drawn automaton with four strong components. *Right:* The local entropy map of the left image.

Example 3. In figure 3, the source image is a binary image of Dr. Helmut Jürgensen.

Note 1. Dr. Jürgensen is my PhD Thesis Supervisor.. The corresponding entropy map is shown beside the original image. There are less areas of complete blackness or whiteness in this image, compared to figure 1, so overall it is more disorderly as one would expect.

Color versions of these example entropy maps can be found at the web address

`http://www.csd.uwo.ca/~meramian/wia99`

and are also printed in [4].

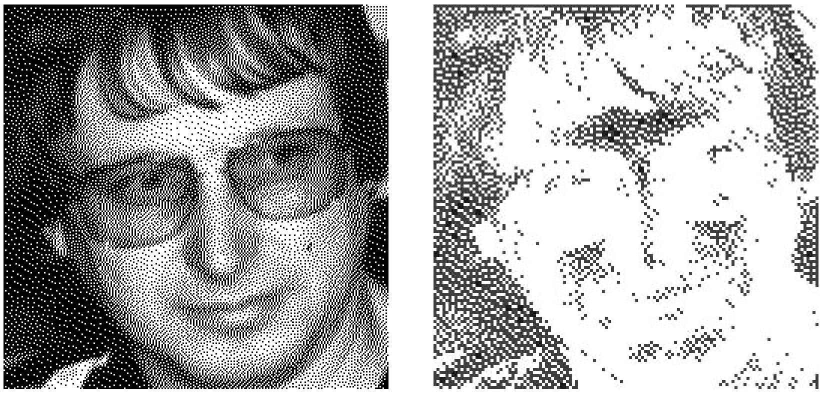


Fig. 3. *Left:* A binary image of Dr. Helmut Jürgensen. *Right:* The local entropy map of the left image.

7 Results and Conclusion

We have shown the first examples of entropy maps of an image created from no information other than the automaton encoding of the image. We plan to apply this technique to a large variety of black and white images and to extend it to greyscale and color images.

References

1. Culik II, K., Kari, J.: Digital Images and Formal Languages. In [9] 599–616.
2. Culik II, K., Kari, J.: Image Compression using Weighted Finite Automata. Comput. and Graphics **17**(3) (1993) 305–313
3. Culik II, K., Valenta, V.: Finite Automata Based Compression of Bi-level and Simple Color Images. Comput. and Graphics **21**(1), (1997) 61–68
4. Eramian, M.: Computing Entropy Maps of Finite-Automaton-Encoded Binary Images. Technical Report #544, Department of Computer Science, University of Western Ontario.
5. Eramian, M., Schincariol, R. A., Mansinha, L., Stockwell, R. G.: Generation of Aquifer Heterogeneity Maps using Two Dimensional Spectral Texture Segmentation Techniques. Mathematical Geology, **31**(3) (1999) 327–348
6. Gerald, C. F., Wheatley, P. O.: Applied Numerical Analysis, Fifth Edition. Addison-Wesley Publishing Company (1994)
7. Jürgensen, H., Staiger, L.: Local Hausdorff Dimension. Acta Informatica **32** (1995) 491–507
8. Pal, N. R., Pal, S. K.: A review on image segmentation techniques. Pattern Recognition **26**(9) (1993) 1277–1294
9. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages Vol. 3, edited by G. Rozenberg and A. Salomaa. Springer-Verlag, Berlin (1997)
10. Seneta, E.: Non-negative Matrices and Markov Chains, second edition. Springer-Verlag, New York (1981)

11. Staiger, L.: Quadtrees and the Hausdorff Dimension of Pictures. In: Geobild '89, Proceedings of the 4^{th} Workshop on Geometrical Problems of Image Processing held in Georgenthal, March 13–17, 1989, edited by A. Hübler, W. Nagel, B. D. Ripley, G. Werner. Akademie-Verlag, Berlin (1989) 173–178
12. Thomas, W.: Automata on Infinite Objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, edited by J. V. Leeuwen. Elsevier, Amsterdam (1994) 133–191
13. Van Gool, L., Dewaele, P., Oosterlinck, A.: Texture Analysis. Comput. Vision Graphics Image Process. **29**(3) (1985) 336–357

Thompson Digraphs: A Characterization*

Dora Giammarresi[1], Jean-Luc Ponty[2], and Derick Wood[3]

[1] Dipartimento di Matematica Applicata e Informatica, Università Ca' Foscari di Venezia, via Torino 155, 30173 Venezia Mestre, Italy.
`dora@dsi.unive.it.`

[2] L.I.F.A.R., Université de Rouen, Faculté des Sciences et des Techniques, Place Émile Blondel, 76821 Mont-Saint-Aignan Cedex, France.
`ponty@dir.univ-rouen.fr`

[3] Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong SAR.
`dwood@cs.ust.hk.`

Abstract. A finite-state machine is called a Thompson machine if it can be constructed from a regular expression using Thompson's construction. We call the underlying digraph of a Thompson machine a Thompson digraph. We establish and prove a characterization of Thompson digraphs. As one application of the characterization, we give an algorithm that generates an equivalent regular expression from a Thompson machine in time linear in the number of states.

1 Introduction

In 1968, Thompson [8] gave an inductive construction of finite-state machines from regular expressions that was motivated by `grep.` The resulting finite-state machines have sizes linear in the sizes of the original expressions. A resurge of interest in the implementation of machines has resulted in some new discoveries about the Thompson construction [2,4].

We characterize the underlying digraphs of the machines resulting from the Thompson construction on empty-free regular expressions (they do not include the empty-set symbol): **Thompson digraphs** and **Thompson machines,** respectively.

First, we characterize Thompson digraphs that are obtained from empty-free, star-free regular expressions. These digraphs are acyclic; therefore, we call them **Thompson dags.** We use Dyck languages defined by source–sink paths in a Thompson digraph in the characterization.

Second, since Thompson digraphs are **hammocks**[1], we can easily find all back edges with a depth-first traversal in linear time. Therefore, in linear time,

* This research was supported under a grant from the Research Grants Council of Hong Kong SAR. It was carried out while the first and second authors were visiting HKUST.

[1] In the literature hammocks are often called *st*-digraphs.

we can determine where the star units occur and we can then transform the digraph into a dag with what we call **star reduction.** The resulting dag is a Thompson dag if and only if the original digraph is a Thompson digraph.

The characterization provides us with a means of generating small regular expressions from some finite-state machines. We first determine whether a given finite-state machine is Thompson and, if it is, we construct a small equivalent regular expression from the machine.

2 Notation and Terminology

We recall the basics of digraphs, finite-state machines and regular expressions and introduce the notation that we use.

A directed graph or **digraph** $G = (V, E)$ consists of a finite set V of vertices and a set E of directed edges of the form (u, v), where u and v are vertices. A **path** is a sequence $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ of edges; it is a **cycle** if $v_0 = v_k$ and $k \geq 1$. A path is a **simple path** if it contains no cycles. A digraph that has no cycles is called a directed acyclic graph or **dag.** The **size of a digraph** is the sum of the number of vertices and number of edges.

We are particularly interested in digraphs that have a single designated **source vertex s** that has no edges entering it, a single designated **sink vertex S** that has no edges exiting it, and each of its vertices occurs on some simple path from the source to the sink. Such digraphs are called **hammocks** and we denote them with a tuple (V, E, s, S).

A **finite-state machine**[2] consists of a finite set Q of states, an input alphabet Σ, a start state $s \in Q$, a final state $f \in Q$ and a transition relation $\delta \subseteq Q \times \Sigma_\lambda \times Q$, where λ denotes the null string and $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. Clearly, we can depict the transition relation of such a machine as an edge-labeled digraph (the labels are symbols from Σ_λ); it is usually called the state or transition digraph of the machine. If we drop the edge labels of a state digraph and ignore multiple edges, we obtain a digraph, the underlying digraph of the machine. The **size of a machine** is the number of its transitions.

Let Σ be an alphabet. Then, we define a regular expression E over Σ inductively as follows:

$E = \emptyset$**,** where $\emptyset$ is the empty-set symbol;
$E = \lambda$**,** where λ is the null-string symbol;
$E = a$**,** where a is in Σ;
$E = (F + G)$**,** where F and G are regular expressions;
$E = (F \cdot G)$**,** where F and G are regular expressions;
$E = (F^*)$**,** where F is a regular expression.

We define the language of E inductively, in the usual way [1,9].

[2] Normally, a finite-state machine is allowed to have more than one final state and, sometimes, more than one start state. The formulation we have chosen is appropriate for the study of Thompson machines.

We say that a regular expression E is **empty free** if E does not contain any appearance of the empty-set symbol. We say that a regular expression is **star free** if it has no Kleene-star subexpression. The size $|E|$ of a regular expression E is the total number of appearances in E of symbols from $\Sigma \cup \{\lambda, \emptyset\}$.

3 Thompson Digraphs

Thompson developed an inductive construction [8] (see Fig. 1) to compile regular expressions into finite-state machines. In Fig. 2, we give the result of the

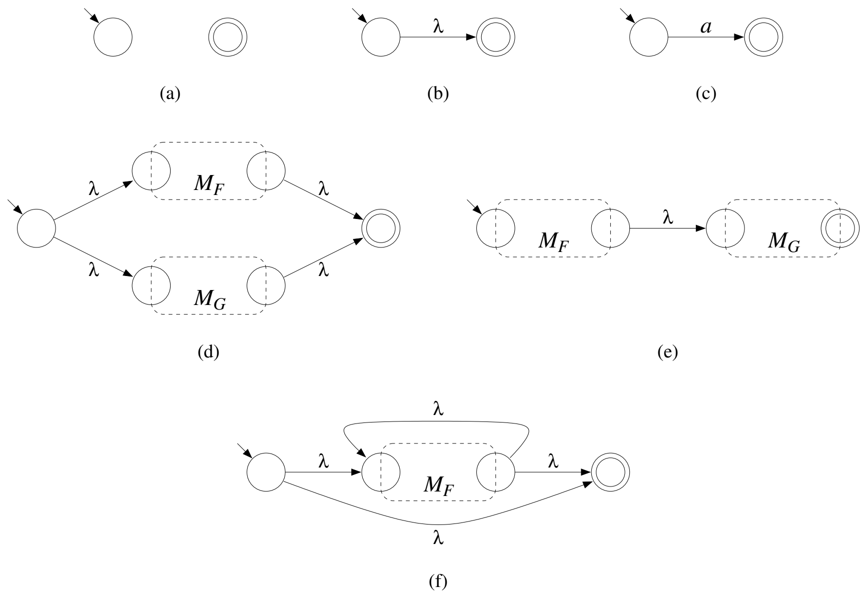


Fig. 1. The Thompson construction. The order of the figures corresponds to the order of the cases in the definition of regular expressions. The finite-state machines correspond to the regular expressions: a. $E = \emptyset$; b. $E = \lambda$; c. $E = a$, $a \in \Sigma$; d. $E = (F + G)$; e. $E = (F \cdot G)$; and f. $E = (F^*)$. When a given regular expression E is empty free, (a) is never used by the Thompson construction. We include (a) for completeness.

Thompson construction on the regular expression $(((a + b)^*) \cdot ((b + \lambda) \cdot a))$.

We define a **Thompson machine** to be a finite-state machine that is obtained by the Thompson construction on an empty-free regular expression and we define a **Thompson digraph** to be the underlying digraph of a Thompson machine. We name the units that the Thompson construction uses to assemble a Thompson machine as follows: the **base units** (Fig. 1(b) and (c)); the **plus unit** (Fig. 1(d)); the **dot unit** (Fig. 1(e)); and the **star unit** (Fig. 1(f)).

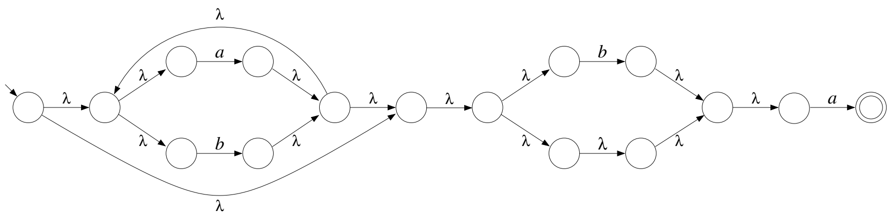


Fig. 2. The result of the Thompson construction on the running example expression $(((a + b)^*) \cdot ((b + \lambda) \cdot a))$.

Notationally, given an empty-free regular expression E, we denote the resulting Thompson machine by M_E^T and, similarly, we denote the corresponding Thompson digraph by H_E^T.

Observe that a Thompson digraph is a hammock[3] and it has exactly the same number of edges as the number of transitions in the original Thompson machine. The Thompson machine of Fig. 2 yields the Thompson digraph of Fig. 3.

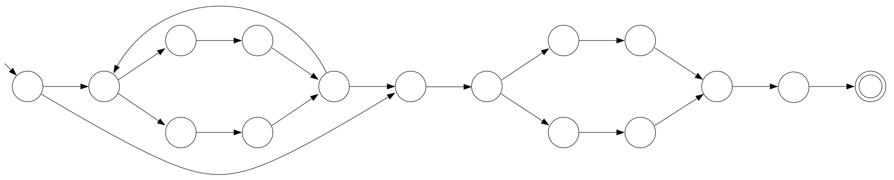


Fig. 3. The Thompson digraph given by the Thompson machine of Fig. 2.

We can obtain a Thompson digraph directly from a regular expression by modifying Thompson's construction appropriately.

4 The Characterization Theorem

Not only is a Thompson digraph a hammock, but also the vertices of a Thompson digraph have indegree and outdegree at most two. We call hammocks that satisfy this additional restriction **two hammocks.**

We characterize the two hammocks that are Thompson digraphs.

We begin with a modification of the usual definition of Dyck strings. For an integer $t \geq 1$, let $\Gamma_t^L = \{[_i : 1 \leq i \leq t\}$, $\Gamma_t^R = \{]_i : 1 \leq i \leq t\}$, $\Gamma_t = \Gamma_t^L \cup \Gamma_t^R$ and $\bar{\Gamma}_t = \Gamma_t \cup \{\bowtie\}$, where $\bowtie$ is a special symbol used to label vertices that

[3] We can establish this observation formally using the inductive construction. If we allow regular expressions to contain the empty-set symbol, then the digraphs of the resulting machines are not necessarily hammocks.

are of indegree and outdegree one. The notion of Dyck string over Γ_t is well known; such a string is well-balanced with respect to the pairing of $[_i$ with $]_i$, for all i, $1 \le i \le t$. We need to allow the pairing to be given by any bijection $\beta : \{1, \ldots, t\} \longrightarrow \{1, \ldots, t\}$ not only by the identity bijection. We call such strings, **b-Dyck strings** and we define them as follows. Given an alphabet Γ_t and a bijection $\beta : \{1, \ldots, t\} \longrightarrow \{1, \ldots, t\}$, we define b-Dyck strings over Γ_t with respect to β inductively as follows: *The null string λ is a b-Dyck string over Γ_t and whenever x and y are b-Dyck strings over Γ_t, xy is a b-Dyck string over Γ_t and $[_i x]_{\beta(i)}$ is a b-Dyck string over Γ_t, for all i, $1 \le i \le t$.*

We now define a fundamental set of strings for Thompson digraphs. Given an alphabet Γ_t and a bijection $\beta : \{1, \ldots, t\} \longrightarrow \{1, \ldots, t\}$, a string x over $\bar{\Gamma}_t$ is a **Thompson string** with respect to β if it satisfies the following three conditions:

1. If we erase all appearances of $\bowtie$ from x, we obtain a b-Dyck string over Γ_t with respect to β.
2. The substring $[_i \,]_j$ does not appear in x, for any i and j, $1 \le i, j \le t$.
3. All maximal-length $\bowtie$-substrings of x have even length.

We develop a characterization of Thompson digraphs by first considering the Thompson digraphs that are obtained from star-free expressions. In this case, we obtain **Thompson dags.** Since Thompson digraphs are two hammocks, we refer to a vertex with indegree i and outdegree j as an **(i,j) vertex.** There are no $(2, 2)$ vertices in Thompson dags. Indeed, $(2, 2)$ vertices are produced only by the Kleene star of a plus subexpression that has the form $((F + G)^*)$. For convenience we assume that the source and the sink vertices have a dummy in-edge and a dummy out-edge, respectively. Then, we have the following necessary condition for a two-hammock H to be a Thompson dag.

Property 1: A Thompson dag has only $(1, 2)$, $(1, 1)$, and $(2, 1)$ vertices, it has an even number of $(1, 1)$ vertices and it has as many $(1, 2)$ vertices as it has $(2, 1)$ vertices.

Definition 1: As a result of Property 1, let the number of $(1, 2)$ vertices (or, equivalently, the number of $(2, 1)$ vertices) be t. We label the $(1, 2)$ vertices arbitrarily and uniquely with $[_i$, $1 \le i \le t$; the $(1, 1)$ vertices with $\bowtie$; and the $(2, 1)$ vertices arbitrarily and uniquely with $]_j$, $1 \le j \le t$. The resulting digraph is a $\bar{\Gamma}_t$-labeled two hammock. If the hammock is a Thompson dag, then the resulting dag is a $\bar{\Gamma}_t$-labeled Thompson dag.

Note that, if $t = 0$, then the two hammock is a line digraph.

We can now state a second necessary condition for a two-hammock H to be a Thompson dag.

Property 2: For each $\bar{\Gamma}_t$-labeled Thompson dag G, there is a bijection $\beta : \{1, \ldots, t\} \longrightarrow \{1, \ldots, t\}$, such that every source–sink path in G spells out a Thompson string.

A Thompson string satisfies three conditions, the first two are easy to check directly. We discuss the checking of the b-Dyck condition of Property 2 when we discuss how to check for Property 3 later in this section.

Lemma 1. *Each Thompson dag satisfies Properties 1 and 2.*

Proof. [4] Let E be a star-free and empty-free regular expression such that H is the Thompson digraph obtained from E. We prove the result by induction on the size of E.

Notice that Properties 1 and 2 are not sufficient for an acyclic two hammock to be a Thompson dag; for example, consider the dag of Fig. 4. Although all source–sink paths in this dag spell out Thompson strings, the dag cannot be obtained by applying the Thompson construction to any star-free regular expression. The reason is that we also need to verify that every path from a

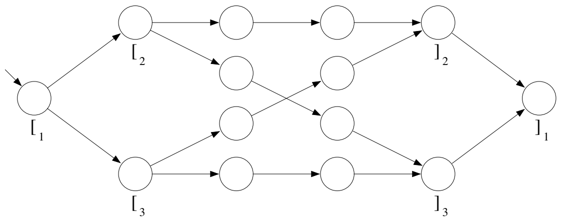


Fig. 4. An example for the insufficiency of Property 2.

$[_i$-vertex passes through the same matching $]_j$-vertex. We express this condition using specific decompositions of b-Dyck strings.

Definition 2: Let H be a $\bar{\Gamma}_t$-labeled two hammock and let p be $(1, 2)$ vertex in H. Then, we define L_p to be the set of strings spelled out on paths from the source vertex to the sink vertex that pass through p.

Property 3: Let H be a $\bar{\Gamma}_t$-labeled Thompson digraph. Then, there is a bijection $\beta : \{1, \ldots, t\} \longrightarrow \{1, \ldots, t\}$ such that, for all $(1, 2)$ vertices p in H and for all strings x in L_p, x can be decomposed into $u[_i v]_{\beta(i)} w$, where $[_i$ is the label of p.

Lemma 2. *Let H be a Thompson dag. Then, H satisfies Property 3.*

We are now in a position to characterize Thompson dags.

Theorem 1. *An acyclic two hammock is a Thompson dag if and only if it satisfies Properties 1, 2 and 3.*

[4] We omit most proofs—they are to be found in the full version of the paper [5].

Proof. We already proved, by Lemmas 1 and 2 that, if H is a Thompson dag, then it satisfies Properties 1, 2 and 3. We now prove the converse. Let H be an acyclic two hammock that satisfies Properties 1, 2 and 3. To demonstrate that H is a Thompson dag, we construct a regular expression E *such that* H *is isomorphic to* H^T_E.

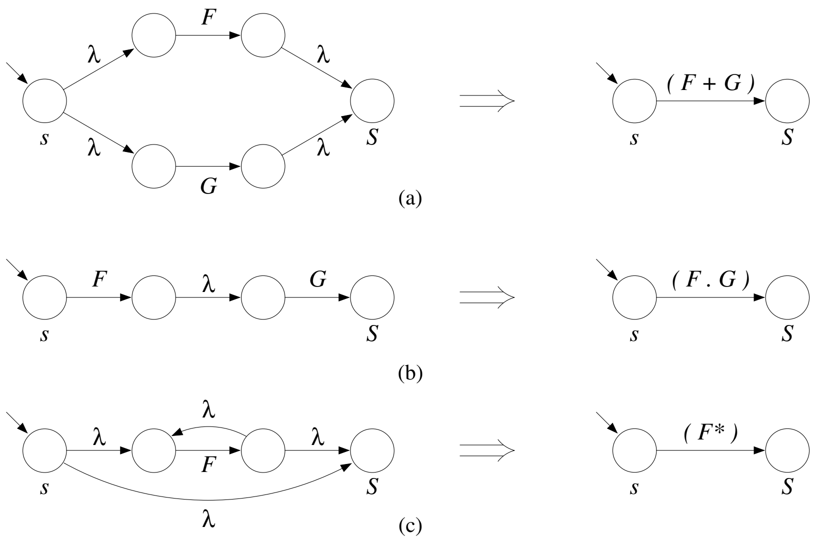


Fig. 5. Three Thompson digraph transformations: a. Replacement of plus unit. b. Replacement of dot unit. c. Replacement of star unit.

We now characterize the two hammocks that are Thompson digraphs. Let H be a two hammock. We perform a depth-first traversal of H starting from the source vertex. It determines a set of **back edges** $B = \{(y_1, x_1), \ldots, (y_k, x_k)\}$, where $k \geq 0$. Notice that when (y_i, x_i) is a back edge, the vertex x_i has indegree two. For, if x_i has indegree one, then there is no simple source–sink path that contains x_i. Similarly, y_i has outdegree two. For, if y_i has outdegree one, then there is no simple source–sink path that contains y_i. We now state a necessary property for a two hammock to be a Thompson digraph.

Property 4: For each back edge (y, x) in a Thompson digraph H, there are two vertices w and z such that w, x, y and z are distinct and edges (w, x), (y, z) and (w, z) are in H.

Based on Property 4, we define a digraph transformation that removes each back edge (y, x) and expands each edge (w, z) into three edges. Thus, if a two hammock has only Thompson-like cycles, the digraph transformation will transform them to give a dag. We define the **star-reduction dag** $\bar{H}$ as follows: for each back edge (y_i, x_i), remove the edge (y_i, x_i) and replace the edge (w_i, z_i) with the three edges (w_i, u_i), (u_i, v_i) and (v_i, z_i), where u_i and v_i are new vertices; see Fig. 6.

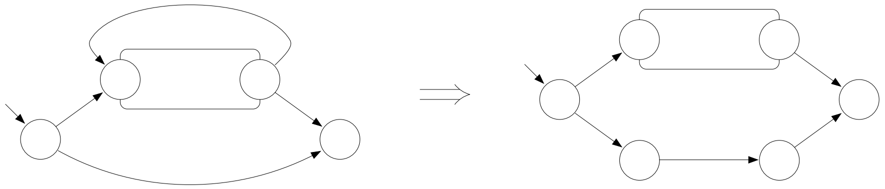


Fig. 6. An illustration of star reduction.

Theorem 2. *A two-hammock H is a Thompson digraph if and only if H satisfies Property 4 and $\bar{H}$ is a Thompson dag.*

Proof. It is easy to verify that, by definition, if H is a Thompson digraph, then each back edge has corresponding forward edges as specified in Property 4 and $\bar{H}$ is a Thompson dag.

Conversely, suppose that we have a two-hammock H that satisfies Property 4 and $\bar{H}$ is a Thompson dag. We can construct a regular expression $\bar{E}$ corresponding to $\bar{H}$ as in the proof of Theorem 1. Then, we can obtain a regular expression E for H from $\bar{E}$ by replacing subexpressions of the form $(F + \lambda)$, produced by star reduction, with (F^*). We omit the formal inductive proof that $H_E^T \cong H$ as it is similar to the proof of Theorem 1.

Given a digraph H, we can verify whether it is a two hammock in time linear in the size of H. Moreover we can check whether it is a Thompson digraph in time linear in the size of H in two steps as follows: First, using a depth-first traversal of H from the source vertex, we detect all back edges and then check whether Property 4 holds. Second, if Property 4 holds, then we apply star reduction to H to give a dag H' and then check by depth-first traversal whether H' is a Thompson dag. We give a more formal description of the nontrivial portion of a Thompson-dag recognition algorithm elsewhere [5].

5 Regular Expressions from Thompson Machines

Once we have confirmed that a given edge-labeled digraph is an edge-labeled Thompson digraph, we can construct a regular expression from it whose size is linear in the size of the given machine and we can do so in linear time.

Theorem 3. *Given a Thompson machine M,we can construct an equivalent regular expression from it in time linear in the size of M.*

Proof. The idea behind the proof of this result is that we, first, parse the underlying edge-labeled Thompson digraph using a depth-first traversal to obtain an expression tree and, second, perform a depth-first traversal of the expression tree to produce a correctly parenthesized regular expression. Clearly, the second step is straightforward and can be implemented to run in time linear in the size of the expression tree. We claim that the size of the expression tree is of the same order as the size of the Thompson digraph. We can construct a parsing algorithm for Thompson digraphs that takes time linear in the size of a digraph [5]. The algorithm assumes that for each starting vertex of a unit, including a base unit, the β map gives the corresponding ending vertex.

It is easy to verify that each edge of H is traversed exactly once by the parsing algorithm; therefore, it takes time linear in the size of H. Moreover its correctness follows because H satisfies Properties 1, 2 and 3.

The parsing algorithm for Thompson digraphs is, essentially, the inverse of the Thompson construction. For, if we take a regular expression E, construct the Thompson machine M_E^T and then apply the parsing algorithm to M_E^T, we obtain a regular expression $\bar{E}$ that is equivalent to E. Note that, although $\bar{E}$ and E are equivalent, $\bar{E}$ may be differently parenthesized and the order of subexpressions of plus units may be different.

6 Concluding Remarks

We have established a characterization of Thompson digraphs that enables us to unambiguously parse such digraphs and reconstruct regular expressions from them that have the same sizes as their digraphs. The interesting fact is that we ignore the transition labels completely in the characterization. The earlier work of Caron and Ziadi [3] gives a second class of machines for which the construction of equivalent regular expressions can be carried out efficiently. Can similar characterizations be established for other inductively constructed finite-state machines? We conjecture that such results hold for Mirkin's construction [6] and the SSS construction [7]. In any case, we conjecture that the finite-state machines given by these constructions can be unambiguously parsed. One interesting problem is whether we can unambiguously parse the machines given by other constructions in the literature including Kleene's original construction.

A tantalizing open problem is to characterize the largest class of finite-state machines that have small expressions easily computable from the machines. A less ambitious goal is to identify nontrivial classes of finite-state machines that yield small expressions.

References

1. A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.
2. A. Brüggemann-Klein and D. Wood. The validation of SGML content models. *Mathematical and Computer Modelling*, 25:73–84, 1997.

3. P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 1998. To appear.
4. D. Giammarresi, J.-L Ponty, and D. Wood. The Glushkov and Thompson constructions: A synthesis. Unpublished manuscript, July 1998.
URL: http://www.cs.ust.hk/tcsc/RR/1998-11.ps.gz.
5. D. Giammarresi, J.-L Ponty, and D. Wood. A characterization of Thompson digraphs. Unpublished manuscript, January 1999.
URL: http://www.cs.ust.hk/tcsc/RR/1999-2.ps.gz.
6. B. G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engineering Cybernetics*, 5:110–116, 1966.
7. S. Sippu and E. Soisalon-Soininen. *Parsing Theory: I Languages and Parsing.* EATCS Monographs on Theoretical Computer Science, Volume 15. Springer-Verlag, New York, NY, 1988.
8. K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.
9. D. Wood. *Theory of Computation.* John Wiley & Sons, Inc., New York, NY, second edition, 1998. In preparation.

Finite Automata Encoding Geometric Figures*

Helmut Jürgensen[1] and Ludwig Staiger[2]

[1] Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B7, and Institut für Informatik, Universität Potsdam, Am Neuen Palais 10, D–14469 Potsdam, Germany; helmut@uwo.ca or helmut@cs.uni-potsdam.de

[2] Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg, Kurt-Mothes-Str. 1, D–06099 Halle (Saale), Germany; staiger@informatik.uni-halle.de

Abstract. Finite automata are used for the encoding and compression of images. For black-and-white images, for instance, using the quad-tree representation, the black points correspond to ω-words defining the corresponding paths in the tree that lead to them. If the ω-language consisting of the set of all these words is accepted by a deterministic finite automaton then the image is said to be encodable as a finite automaton. For grey-level images and colour images similar representations by automata are in use.

In this paper we address the question of which images can be encoded as finite automata with full infinite precision. In applications, of course, the image would be given and rendered at some finite resolution – this amounts to considering a set of finite prefixes of the ω-language – and the features in the image would be approximations of the features in the infinite precision rendering.

We focus on the case of black-and-white images – geometrical figures, to be precise – but treat this case in a d-dimensional setting, where d is any positive integer. We show that among all polygons in d-dimensional space those with rational corner points are encodable as finite automata. In the course of proving this we show that the set of images encodable as finite automata is closed under rational affine transformations.

Several simple properties of images encodable as finite automata are consequences of this result. Finally we show that many simple geometric figures such as circles and parabolas are not encodable as finite automata.

1 Introduction

Finite automata are widely used as a means for describing certain fractals (see [1, 2,7]). Usually, the investigation of automaton-generated fractals starts from the underlying automaton and aims at a description of the image or the calculation

* The research reported in this paper was partially supported by the Natural Sciences and Engineering Research Council of Canada, Grant OGP0000243. After completion of this paper we succeeded in strengthening some of the results. A full version of this paper, which includes all proofs and the recent results, is in preparation [6].

of some of its parameters like density, dimension or measure (see [5,4]). Less is known about the converse direction, that is, starting from a class of images to ask whether they are generated by automata or, if so, to describe these automata. Some structural properties of images generated by finite automata can be derived from the structure of the ω-languages accepted by the automata. Finite-automaton generated images turn out to have specific shapes (see e. g. [1, 7]).

We focus on d-dimensional black-and-white images. Using their representation as infinite (ordered) trees with a branching of up to 2^d – in the case of $d = 2$ these are quad-trees – the black points correspond to the infinite branches in these trees. Hence an image would be represented by the ω-language describing these branches. An image is *encodable as (or definable by) a finite automaton* if its ω-language is accepted by that automaton, that is, if that ω-language is regular (see [10]). The cases of grey-level or colour images would require additional parameters.

The encoding of an image as an automaton represents the image at an infinite resolution. Sampling or rendering the image at a bounded resolution corresponds to running the automaton for a bounded time only. These connections are exploited, for example, in an automaton-based image compression procedure (see [3]).

We address the question of which images are encodable as finite automata. In particular, we consider polygons and simplexes in d-dimensional Euclidean space, that is, convex hulls of finite sets of points.

The main theorems of this paper state that a d-dimensional simplex is definable by a finite automaton if it is the convex hull of a finite set of points with rational coordinates, and a polygon is definable by a finite automaton if and only if its corner points are rational. This result is independent of the base chosen for the number representation. The set of images definable by finite automata being closed under union, projection, inverse projection and, essentially, also difference,[1] it turns out that the class of geometrical figures definable by finite automata is quite rich.

One of the main tools for proving this result is the following property of images encodable as finite automata: The set of these images is closed under rational affine transformations, that is, transformations of the form $\mathfrak{y} = A\mathfrak{x} + \mathfrak{b}$ with only rational numbers as entries of the transformation matrix A and the translation vector $\mathfrak{b}$.

From closure properties of the set of regular ω-languages and these results, one can determine further interesting classes of simple geometrical figures encodable as finite automata. On the other hand, some very simple geometrical figures like circles or parabolas cannot be encoded as finite automata. For image compression by automata this implies that such figures will, of necessity, be approximated by simplexes sampled at some bounded resolution.

[1] We consider figures that are bounded and closed in Euclidean space. Therefore, *difference* here means the closure of the set theoretical difference.

2 Notation

The symbols $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ denote the sets of non-negative integers, integers, rational and real numbers, respectively. An alphabet is a finite and non-empty set. For an alphabet X, X^* and X^ω denote the sets of finite and right-infinite words over X, respectively. For a word $w \in X^*$, $|w|$ is its length. Right-infinite words are referred to as ω-words in the sequel. An ω-language is a set of ω-words. An ω-language is regular if it is accepted by a finite automaton (see [10] for the relevant background and references).

For any alphabet Y and any positive integer d, let $[Y, d]$ denote the d-fold Cartesian product

$$[Y, d] = \underbrace{Y \times \ldots \times Y}_{d \text{ times}}.$$

For $y = (y_1, \ldots, y_d) \in [Y, d]$ and an integer i with $1 \le i \le d$, the i-th projection of y is $\operatorname{proj}_i y = y_i$.

For the representation of real numbers, we fix a base $r \in \mathbb{N}$ with $r \ge 2$. Then the set $Y = \{0, 1, \ldots, r-1\}$ is considered as the set of r-ary number symbols. Every real number in the closed interval $[0,1] = \{x \mid 0 \le x \le 1\}$ has a base-r representation of the form $0.\alpha$ where $\alpha \in Y^\omega$. In particular, a finite representation of a rational number can be padded by an infinite sequence of the symbol 0. Conversely, every ω-word α over Y denotes a unique real number $\nu_r(\alpha)$ in the interval $[0,1]$, represented by $0.\alpha$. It is well-known that the mapping from representations of numbers to their values is not injective.

Let d be a positive integer. To specify points in the closed d-dimensional unit cube $[0,1]^d$ we use ω-words over the alphabet $X = [Y, d]$. For $\xi = x_1 x_2 \ldots \in X^\omega$ and an integer i with $1 \le i \le d$, the i-th projection of ξ is the ω-word

$$\operatorname{proj}_i \xi = \operatorname{proj}_i x_1 \operatorname{proj}_i x_2 \cdots$$

obtained from the i-th projections of the symbols of x. The point $\nu_r(\xi)$ in $[0,1]^d$ defined by ξ has, as coordinates, the values of the numbers represented by the projections of ξ.

We generalize this concept of projection to multiple coordinates. Consider $y = (y_1, \ldots, y_d) \in X = [Y, d]$, $k \in \mathbb{N}$, $k > 0$, and a k-tuple $\mathfrak{i} = (i_1, \ldots, i_k)$ of integers in $\{1, \ldots, d\}$. Then $\operatorname{proj}_{\mathfrak{i}} y = (y_{i_1}, \ldots, y_{i_k}) \in [Y, k]$. For $\xi = x_1 x_2 \cdots \in X^\omega$, the projection $\operatorname{proj}_{\mathfrak{i}} \xi$ is the ω-word

$$\operatorname{proj}_{\mathfrak{i}} x_1 \operatorname{proj}_{\mathfrak{i}} x_2 \cdots$$

in $[Y, k]^\omega$ and its value $\nu_r(\operatorname{proj}_{\mathfrak{i}} \xi)$ is a point in $[0,1]^k$. Let $\operatorname{pr}_{\mathfrak{i}}$ denote the corresponding projection of $[0,1]^d$ into $[0,1]^k$. The following diagram is commutative.

$$\begin{tikzcd}
[Y,d]^\omega \arrow[r, "\operatorname{proj}_{\mathfrak{i}}"] \arrow[d, "\nu_r"] & {[Y,k]^\omega} \arrow[d, "\nu_r"] \\
{[0,1]^d} \arrow[r, "\operatorname{pr}_{\mathfrak{i}}"] & {[0,1]^k}
\end{tikzcd}$$

The mapping $\operatorname{proj}_{\mathfrak{i}}$ and its inverse preserve regularity of ω-languages.

Let $\eta_1, \ldots, \eta_d \in Y^\omega$. By slight abuse of notation we write $(\eta_1, \ldots, \eta_d)$ to denote the ω-word $\xi \in [Y, d]^\omega$ such that $\operatorname{proj}_i \xi = \eta_i$ for $i = 1, \ldots, d$.

On X^ω one defines an ultrametric ϱ by

$$\varrho(\zeta, \xi) = \inf\{r^{-|w|} \mid w \text{ is a common prefix of } \zeta \text{ and } \xi\}.$$

Since X is finite, the space (X^ω, ϱ) is a compact metric space. Moreover, the mapping ν_r of X^ω onto $[0,1]^d$ is continuous.

3 Rational Affine Transformations

Consider a function $\varphi : \mathbb{R}^d \to \mathbb{R}$ and an ω-language F over X. The function φ is said to *describe the ω-language F* if F is the largest ω-language such that $\nu_r(F)$ is the set of all solutions in $[0,1]^d$ of the equation

$$\varphi(x_1, \ldots, x_d) = 0,$$

that is,

$$F = \nu_r^{-1}\left(\{(x_1, \ldots, x_d) \mid \varphi(x_1, \ldots, x_d) = 0, 0 \le x_i \le 1 \text{ for } i = 1, \ldots, d\}\right).$$

We write F_φ to denote the ω-language described by φ. The set F_φ contains *all* base-r representations of all solutions in $[0,1]^d$ of the equation above.

The following lemma plays a fundamental rôle in some of the proofs:

Lemma 1. *Let $\varphi : \mathbb{R}^d \to \mathbb{R}$ be a function, $c_i \in \{-1, +1\}$, $1 \le i \le d$, and $c \in \mathbb{Z}$ such that*

$$\varphi(x_1, \ldots, x_d) = c_1 x_1 + \cdots + c_d x_d + c.$$

Then F_φ is regular and closed.

We list a few immediate consequences. As is well-known every rational number of the form k/r^l has two base-r representations. Thus a point in d-dimensional space $\mathbb{R}^d$ may have up to 2^d representations. A typical complication arises from the fact that, due to those multiple representations, for $F, F' \subseteq X^\omega$, the sets $\nu_r(F) \cap \nu_r(F')$ and $\nu_r(F \cap F')$ might not be equal. For example, with $d = 1$, $r = 2$, $F = \{1000\cdots\}$ and $F' = \{01111\cdots\}$ one has $\nu_r(F) = \nu_r(F') = \{\frac{1}{2}\}$ whereas $\nu_r(F \cap F') = \emptyset$. However, for any $F, F' \subseteq X^\omega$, one has

$$\nu_r(F) \cap \nu_r(F') = \nu_r\left(\nu_r^{-1}(\nu_r(F)) \cap F'\right).$$

One is, therefore, led to work with *full representations,* that is, with ω-languages F satisfying $F = \nu_r^{-1}(\nu_r(F))$.

Proposition 1 *The ω-languages*

$$E^{(2d)} = \left\{\xi \mid \xi \in [Y, 2d]^\omega \textit{ with } \nu_r(\operatorname{proj}_i \xi) = \nu_r(\operatorname{proj}_{i+d} \xi) \textit{ for } i = 1, \ldots d\right\}$$

and

$$E^{[m]} = \left\{\xi \mid \xi \in [Y, m+1]^\omega \textit{ and } \nu_r(\operatorname{proj}_2 \xi) = \nu_r(\operatorname{proj}_{i+1} \xi) \textit{ for } i = 1, \ldots, m\right\}$$

are regular.

As a consequence, moving from a regular representation to the corresponding full representation preserves regularity.

Proposition 2 *Let F be an ω-language over $X = [Y, d]$. If F is regular then also $\nu_r^{-1}(\nu_r(F))$ is regular.*

We now include integer and, consequently, also rational coefficients.

Proposition 3 *Consider a function $\varphi : \mathbb{R}^2 \to \mathbb{R}$ such that $\varphi(x_1, x_2) = x_1 - mx_2$ for some $m \in \mathbb{Z}$. Then F_φ is regular.*

Exploiting the proof techniques of of Propositions 1–3 one can extend Lemma 1 to rational coefficients.

Lemma 2. *Let $\varphi : \mathbb{R}^d \to \mathbb{R}$ be a function, $c_i, c \in \mathbb{Z},\ 1 \le i \le d$, such that*

$$\varphi(x_1, \ldots, x_d) = c_1x_1 + \cdots + c_dx_d + c.$$

Then F_φ is regular and closed.

An affine transformation of $\mathbb{R}^d$ into $\mathbb{R}^k$ is given by an equation of the form $\mathfrak{y} = A\mathfrak{x} + \mathfrak{b}$ where $\mathfrak{y}$ and $\mathfrak{b}$ are $1 \times k$-vectors, $\mathfrak{x}$ is a $1 \times d$-vector and A is a $k \times d$-matrix. An affine transformation is said to be *rational* if the entries of A and $\mathfrak{b}$ are rational.

Theorem 1. *Let $\Psi : \mathbb{R}^d \to \mathbb{R}^k$ be a rational affine transformation and let $\Gamma(\Psi) \subseteq \mathbb{R}^{d+k}$ be its graph. Then the ω-language $F_\Psi = {\nu_r}^{-1}\big(\Gamma(\Psi) \cap [0,1]^{d+k}\big)$ is regular.*

From Theorem 1 one concludes that rational affine transformations and their inverses preserve regularity.

Theorem 2. *Let $\Psi : \mathbb{R}^d \to \mathbb{R}^k$ and $\Phi : \mathbb{R}^k \to \mathbb{R}^d$ be rational affine transformations and let $F \subseteq X^\omega$ be regular. Then both*

$$\nu_r^{-1}\left(\Psi(\nu_r(F)) \cap [0,1]^k\right) \ \textit{and}\ \nu_r^{-1}\left(\Phi^{-1}(\nu_r(F)) \cap [0,1]^k\right)$$

are regular ω-languages.

4 Simple Geometric Figures

A point in $[0,1]^d$ is said to be rational if all its coordinates are rational; it is said to be nearly rational if at least $d-1$ of its coordinates are rational. A simplex in $[0,1]^d$ is the convex hull of a finite set of points in $[0,1]^d$. A simplex in $[0,1]^d$ is said to be *rational* if it is the convex hull of finitely many rational points. Using Theorem 2, one obtains a sufficient condition on the encodability of simplexes as finite automata.

Theorem 3. *A simplex in $[0,1]^d$ is encodable as a finite automaton if it is rational.*

Using the closure properties of regular ω-languages and taking into account that we need to work with full representations one realizes that set of simple geometric figures definable by finite automata is quite large.

Since the closure and the boundary of regular ω-language are again regular, we obtain the following closure property of the family of images definable by finite automata

Proposition 4 *Let $M \subseteq [0,1]^d$ be encodable as a finite automaton. Then both the closure $\overline{M}$ and the boundary ∂M of M are encodable as finite automata.*

We obtain a characterization of polygons encodable as finite automata.

Theorem 4. *A polygon in $[0,1]^d$ is encodable as a finite automaton if and only if its corner points are rational.*

In the course of proving Theorem 4 one derives several criteria for the encodability of line sets in the unit interval as automata.

Proposition 5 *Let $M \subseteq [0,1]^d$ be encodable as a finite automaton. If M is non-empty then it contains a rational point. If M is countable, then all points in M are rational.*

Lemma 3. *Let I be finite or denumerable index set and, for $i \in I$, let $a_i, b_i \in [0,1]$ with $a_i \leq b_i$; R_i be an interval of the form (a_i, b_i), $[a_i, b_i)$, $(a_i, b_i]$ assuming $a_i \neq b_i$, or $[a_i, b_i]$. If, for $i, j \in I$ with $i \neq j$, the intervals R_i and R_j are disjoint and the set $M = \bigcup_{i \in I} R_i$ is encodable as a finite automaton then a_i and b_i are rational for all $i \in I$.*

5 Images That Are Not Encodable as Finite Automata

There are many images that are not encodable as finite automata. Proposition 5 states a necessary condition for an image to be encodable as a finite automaton. Here we state and apply other necessary conditions.

Proposition 6 *If a smooth non-constant curve $M \subseteq [0,1]^d$ is encodable as a finite automaton then every nearly rational point on the curve is rational.*

From Proposition 6 one finds many simple examples of (two-dimensional) images not encodable as finite automata, for instance:

Example 1 The parabola $f(a) = a^2$ with $0 \leq a \leq 1$ is not encodable as a finite automaton because it contains the non-rational point $\left(1/\sqrt{2}, 1/2\right)$ which has one irrational and one rational coordinate.

The next example uses also Theorem 2 in order to prove the nonencodability.[2]

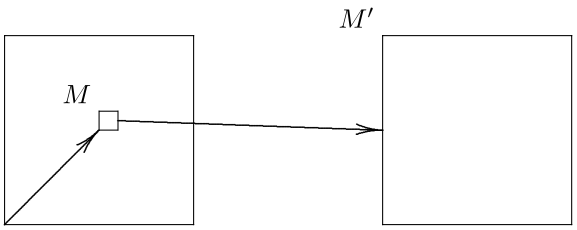
[2] We are grateful to one of the referees for providing us with this simple instructive example.

Example 2 Consider the hyperbola $g(x) = 1/(x+1)$. Every point in $\Gamma(g)$ with one rational coordinate is rational. Now transform $\Gamma(g)$ via the rational affine mapping given by $A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ and $\mathfrak{b} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$. The image is $\Gamma(g')$ where $g'(x) = x^2/(1+x)$ which contains the point $(\frac{1+\sqrt{13}}{6}, \frac{1}{3})$ with exactly one rational coordinate. Thus g is not encodable as a finite automaton.

Another necessary condition is implicit in the following lemma.

Lemma 4. *Let $f : [0,1] \to [0,1]$ be a continuous function, differentiable at a point $a_0 \in [0,1]$ for which $f'(a_0)$ is irrational. Then the graph $\Gamma(f)$ is not encodable as a finite automaton.*

For the proof one uses the following *zoom-in property* of images encodable as finite automata.



$$M \overset{\nu_r}{\leftarrow} F \cap wX^\omega \qquad\qquad M' \overset{\nu_r}{\leftarrow} w^{[-1]}F = \{\xi \mid w\xi \in F\}$$

For $F \subseteq X^\omega$ and $w \in X^*$, let $w^{[-1]}F = \{\xi \mid w\xi \in F\}$. The set $\{w^{[-1]}F \mid w \in X^*\}$ is finite for regular F. The converse is not true in general; see [9] for details. As a consequence, the number of different images obtainable as zoom-ins is finite if the image itself is encodable as a finite automaton.

Corollary 1 *Let $f : [0,1] \to [0,1]$ be a continuously differentiable function with a non-constant derivative. Then the graph $\Gamma(f)$ is not encodable as a finite automaton.*

This corollary explains, in addition to Examples 1 and 2 also the following example.

Example 3 No circle is encodable as a finite automaton.

References

1. J. Berstel, M. Morcrette: Compact Representations of Patterns by Finite Automata. In: *Proc. Pixim '89,* Hermes, Paris, 1989, 387–402.
2. K. Čulik II, S. Dube: Rational and Affine Expressions for Image Description. *Discrete Appl. Math.* **41** (1993), 85–120.
3. K. Čulik II, J. Kari: Digital Images and Formal Languages. In [8], 599–616.
4. M. Eramian: Computing Entropy Maps of Bi-Level Images. *These Proceedings.*

5. H. Jürgensen, L. Staiger: Local Hausdorff Dimension. *Acta Informatica* **32** (1995), 491–507.
6. H. Jürgensen, L. Staiger, H. Yamasaki: Encoding Figures by Automata. *Manuscript, 1999.*
7. W. Merzenich, L. Staiger: Fractals, Dimension, and Formal Languages. *RAIRO–Inform. Théor.* **28** (1994), 361–386.
8. G. Rozenberg, A. Salomaa (eds.): *Handbook of Formal Languages,* Vol. 3, Springer-Verlag, Berlin, 1997.
9. L. Staiger, Finite-state ω-languages. J. Comput. System Sci. 27(1983) 3, 434–448.
10. L. Staiger: ω-Languages. In [8], 339–387.

Compressed Storage of Sparse Finite-State Transducers

George Anton Kiraz

Bell Labs – Lucent Technologies
Room 2D-430, 700 Mountain Ave.
Murray Hill, NJ 07974
gkiraz@research.bell-labs.com

Bees... by virtue of a certain geometrical forethought... know that the hexagon is greater than the square and the triangle and will hold more honey for the same expenditure of material.

Pappus of Alexandria (260 BC – ?)
Mathematical Collection

Abstract. This paper presents an eclectic approach for compressing weighted finite-state automata and transducers, with minimal impact on performance. The approach is eclectic in the sense that various complementary methods have been employed: row-indexed storage of sparse matrices, dictionary compression, bit manipulation, and lossless omission of data. The compression rate is over 83% with respect to the current Bell Labs finite-state library.

1 Introduction

Regular languages are the least expressive among the family of formal languages; hence, their computational counterparts, viz. finite-state automata (FSAs), require the least computational power in terms of space and time complexity. This, as well as other advantages discussed below, makes FSAs very attractive for solving problems in a wide range of computational domains, including switching theory, testing circuits, pattern matching, speech and handwriting recognition, optical character recognition, encryption, data compression and indexing, not to mention a wide range of problems in computational linguistics. (For a recent collection of papers on language processing using automata theory, see (Roche and Schabes, 1997).)

There are other advantages to using FSAs, as well as finite-state transducers (FSTs). Firstly, they are easy to implement. A simple automaton can be represented by a matrix, though this is not necessarily the best solution spacewise as we shall see. Secondly, they are fast to traverse, especially in the case of deterministic devices. Thirdly, and may be most importantly, transducers are bidirectional, e.g., a letter-to-sound rule compiled into an FST becomes a sound-to-letter rule by merely inverting the machine. Finally, FSAs and FSTs are mathematically elegant since they are closed under various useful operations. FSAs are closed under concatenation, union, intersection, difference and Kleene star. FSTs are closed under the same operations (except intersection and difference under which only the subclass of ϵ-free FSTs are closed – ϵ represents the empty string). Transducers are also closed under composition.

Despite all these niceties, large scale implementations of speech and language problems using this technology result in notoriously large machines, posing serious obstacles in terms of time and space complexity. When a finite-state network becomes unmanageable, even with current computer machinary, the original network is usually divided into smaller modules that can be put together at run-time using operations under which FSAs and FSTs are closed. Even with this 'trick', the modules themselves still require massive storage. Table 1 gives the number of modules, and their storage size in megabytes, for the text analysis components of the Bell Labs multi-lingual text-to-speech (TTS) system. The size of these modules becomes a serious hurdle when the system is to be imported into a special-purpose hardware device with limited memory.

Table 1. Storage requirement for letter-to-sound rules in Bell Labs multilingual TTS system

Language	Modules	Size (MB)
German	49	26.6
French	52	30.0
Mandarin	51	39.0

This work presents an eclectic approach for compressing FSAs and FSTs, with minimal impact, if any, on performance when a caching mechanism is activated. The approach adopted here is eclectic in the sense that various complementary methods have been employed: row-indexed storage of sparse matrices, dictionary compression, bit manipulation, and lossless omission of data.

The implementation described herein builds on an object-oriented finite-state machinary library, designed originally by M. Riley and F. Pereira.[1] Data provided throughout the paper is based on weighted finite-state transducers used in the language analysis components of the Bell Labs multilingual TTS system (Sproat, 1997).

Section 2 discusses the characteristics of automata and transducers used in language and speech applications. Section 3 presents the compression approach employed here. Section 4 gives further eclectic compression methods. Finally, section 5 gives some results and brief concluding remarks.

[1] This library was implemented at AT&T Bell Labs. After the trivestiture of the company in 1996, the library was inherited, and developed further independently, by both AT&T Research (Mohri, Pereira, and Riley, 1998) and Lucent Technologies' Bell Labs. The AT&T version is available online at `http://www.research.att.com/sw/tools/fsm` [URL checked July 1, 1999].

2 On Speech and Language Transducers

Before embarking on the task at hand, it is crucial to study some of the characteristics of speech and language FSTs.

2.1 Lexical vs. Rule Machines

Speech and Language automata tend to be of two types. Lexical-based automata describe a set of words, morphemes, etc. Fig. 1 gives an automaton for a small English lexicon representing the words: /book/, /hook/, /move/, and /model/. Final states mark the end of a lexical entry. Note that entries which share prefixes (in the formal sense), such as "mo" in /move/ and /model/, share the same transitions for the prefix. The same holds for suffixes.

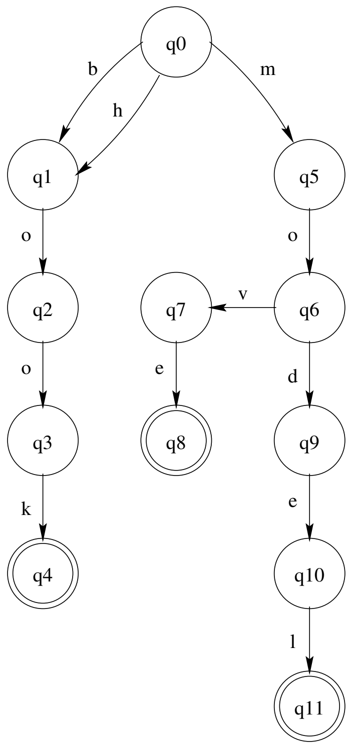


Fig. 1. Lexical representation by automata.

Rule-based FSTs represent a transformational rule. As a way of illustration, consider the derivation of /moving/ from the lexical morphemes /move/ and /ing/. Note that the [e] in /move/ is deleted once the two morphemes join. This final-*e*-deletion rule takes place following a consonant ([v] is this case) and preceding the suffix /ing/. The FST representation of this rule is depicted in Fig. 2.1. The transition *:* from a particular state represents any mapping other than those explicitly shown to be leaving that state. Given a particular input, the transducer remains in state 0 until it scans a 'v' on its input tape, upon which it will output 'v' and move to state 1. Once in state 1, the left context of the rule has been detected.

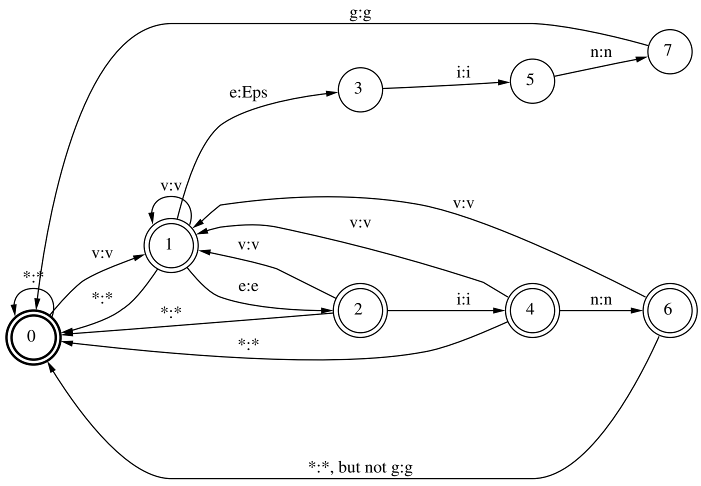


Fig. 2. Transducer for deleting 'e' of /move/ in /moving/. Unless otherwise specified, the transition *:* stands for on all other symbols not specified in the respective state. 'Eps' represents ϵ.

When in state 1 and faced with an 'e' (the symbol in question for this rule), the transducer has two options: (i) to map the 'e' to an ϵ, i.e. apply the rule, and move to state 3, or (ii) retain the 'e' on the output and move to state 2. In the former case, one expects to see the right context since the rule was applied; the transition on i:i to state 5 and the subsequent transition back to state 0 fulfill this task. In the latter case, one expects to see anything *except* the suffix /ing/. Hence, it is not possible to scan /ing/ from state 2.

In large-scale applications, lexical-based FSTs tend to be much larger than rule based ones. They also tend to be more sparse (see §2.3). This observation has an impact on the compression results that can be achieved.

2.2 Parallel vs. Serial Architecture

There are two models for putting FSTs together in a regular grammar environment: parallel and serial. In the parallel model, each regular rule (or lexicon) is compiled into an FST using a compiler similar to that of

(Karttunen and Beesley, 1992). The entire grammar is then taken as the intersection of all such FSTs. Fig. 3(a) illustrates this architecture. Note that since ϵ-containing FSTs are not closed under intersection, this model is only valid with ϵ-free grammars.

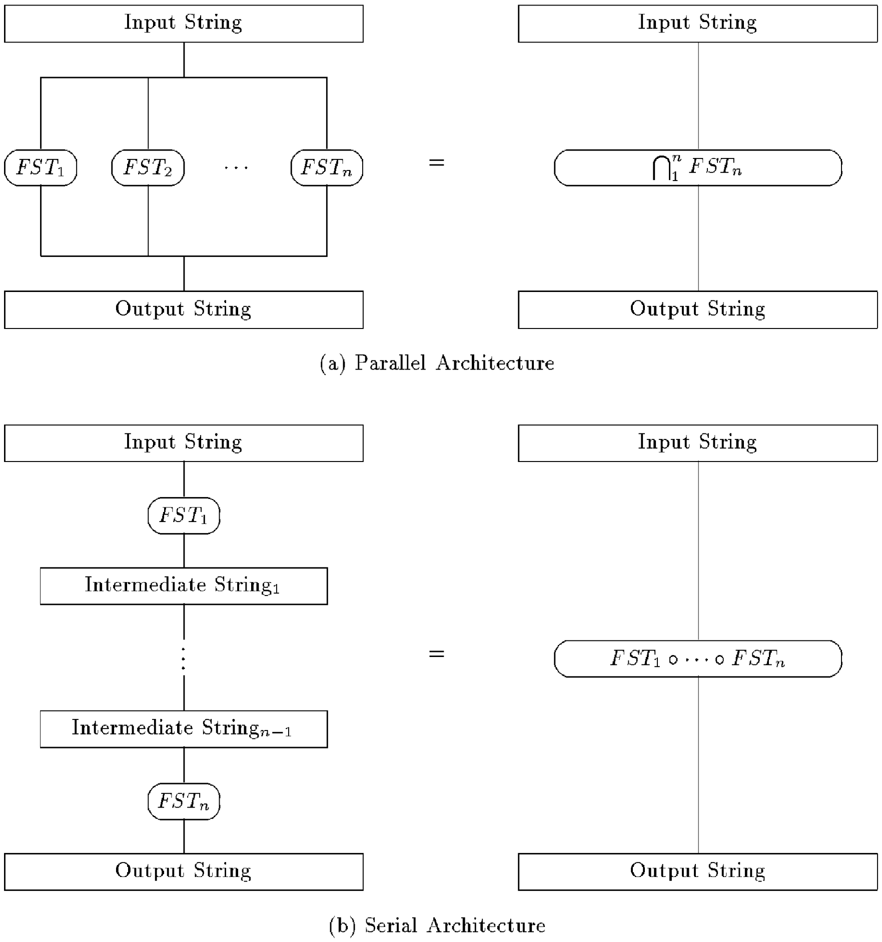


Fig. 3. Parallel and serial architectures for putting FSTs together.

The serial model, which is used to build the FSTs described here, allows for ϵ-containing rules. As in the parallel model, each rule (or lexicon) is compiled into an FST, albeit using a different algorithm (Kaplan and Kay, 1994; Mohri and Sproat, 1996). Under this model, the entire grammar is represented by the serial composition of all such FSTs. Fig. 3(b) illustrates this approach. In both models intermediate machines tend to blow up in size.

The language analysis FSTs at hand represent hundreds of rules as well as various lexical transducers. Composing all rules and lexica into one FST at compile-time is not feasible space-wise since intermediate machines explode in size. Alternatively, composing the input with hundreds of compositions at run-time is computationally expensive time-wise. A middle solution, which is used in the Bell Labs TTS architecture, is to divide rules and lexica into subsets and compile the members of each subset into a module. The input can then be composed with these modules at run-time. Hence, the size of the intermediate machines will always be in the order of the input size. This is of course possible because composition is associative.

2.3 Sparsity

Let $A = (Q, S, q_0, \delta, F)$ be a (nondeterministic) finite-state automaton where Q is a finite set of states, S is a finite set of symbols representing the alphabet, $q_0 \in Q$ is the initial state, $\delta\colon Q \times S \to 2^Q$ (where 2^Q denotes the power set of Q) is the transition function, and $F \subseteq Q$ is a set of final states. The sparsity rate of A is

$$\mathit{Sparsity}(A) = 1 - \frac{|\delta|}{|Q| \times |S|} \tag{1}$$

Our language analysis machines show that FSTs that represent natural language are highly sparse, with lexical FSTs being more sparse than rule based ones. For instance, a lexical transducer of 36,668 English words, with an alphabet size of 112 symbols (including orthographic, phonetic and other grammatical labels) exhibits a sparsity of 98.48%. The English orthographic rules of (Ritchie et al., 1992, §D.1) produce a machine with a sparsity of 93.16%. The sparsity of the stress rules in (Halle and Keyser, 1971, p. 10) is 40.84%. Languages with larger alphabets tend to be notoriously more sparse. One of our Mandarin FSTs, for example, has a sparsity rate of 99.96% due to the fact that Mandarin employs an inventory of more than 17,000 symbols.

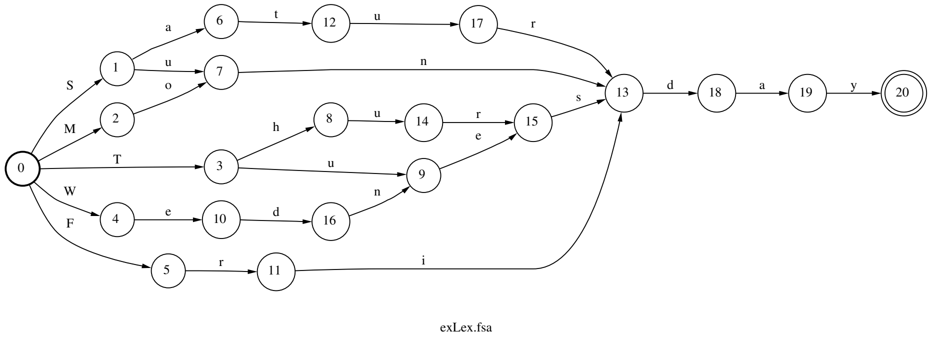
Many mathematical models that give rise to sparse data result in uniform patterns of data. Unfortunately, the sparse data that result from natural language FSTs lack uniformity in their pattern. Hence, one cannot make use of special coding algorithms that are applicable to uniform data.

3 Compressed Storage of Sparse Automata

As the transition matrix representation of FSAs is sparse, it is natural to explore techniques for storing general sparse matrices. This section presents such a technique for representing an accepting automaton and weighted finite-state transducers.

3.1 Data Structures and Storage Schemes

The simplest representation of an automaton is a transition matrix. Rows in the matrix denote states, while columns denote the entire alphabet. An entry on row $q \in Q$ and column $s \in S$ gives the set of next states that can be reached from state q on the symbol s. (In the case of transducers, each entry is a set of pairs (p, o), with p denoting the destination state and o the output symbol. In the case of weighted devices, a weight w is added.)



(a)

State	F	M	S	T	W	a	d	e	h	i	n	o	r	s	t	u	y
0	5	2	1	3	4												
1						6										7	
2												7					
3									8							9	
4								10									
⋮																	
16											9						
17													13				
18						19											
19																	20
20																	

(b)

Fig. 4. An automaton for the days of the week with its transition matrix. The final state (20) is marked in bold

Consider an automaton that accepts the strings representing the days of the week. Fig. 4(a) gives the transition diagram, with the corresponding transition matrix in Fig. 4(b). The advantage of this representation is its random access: once in a particular state is faced with an input symbol, it takes one lookup operation into the table to determine the destination state. However, when matrices become notoriously sparse, especially with the sparsity figures mentioned in section 2.3, such a representation will be deemed infeasible from a space complexity point of view. The matrix in Fig. 4(b) requires $|Q| \times |S|$ storage space. Its sparsity is $1 - \frac{26}{21 \times 122} = 98.985\%$ (122 is the total number of symbols in the English text-analysis system).

3.2 Row-Indexed Storage

Instead of a matrix, an accepting automaton is represented by three arrays. The first array, A, stores the entries from the full matrix representation. For example, the entries from Fig. 4(b) are stored in array A of Fig. 3.2 row-wise. The second array, A_q, stores the indices in A where the entries for each state begins. For example, the entries of state 1 begin in position 5 in A; hence, $A_q[1] = 5$ (we use this notation to state that element 1 in array A_q is 5). In a similar fashion, the entries of state 3 begin in position 8 in A; hence, $A_q[3] = 8$. Finally, for each entry in A, the third array A_s stores the corresponding symbol from the header of Fig. 4(b). For example, the first entry in A, viz. 5, is in the column marked F in Fig. 4(b); hence, $A_s[0] = \mathrm{F}$.

index k	0	1	2	3	4	5	6	7	8	9	10	$\cdots$	22	23	24	25
A	5	2	1	3	4	6	7	7	8	9	10	$\cdots$	9	13	19	20
A_q	0	5	7	8	10	$\cdots$	22	23	24	25						
A_s	F	M	S	T	W	a	u	o	h	u	e	$\cdots$	n	r	a	y

Fig. 5. Row-indexed representation of a sparse automaton.

In terms of space complexity, the length of A equals the number of arcs in the machine, $|\delta|$; the length of A_q equals the number of states, $|Q|$; and the length of A_s equals that of A. Hence, the space complexity of this representation is $2|\delta| + |Q|$, where $|\delta|$ is the number of non-zero entries in the original matrix and $|Q|$ is the number of states. Note that the number of symbols employed, $|S|$, does not feature in this complexity.

As for time complexity, consider the following algorithm which returns the destination state given the current state q and the input symbol s.

```
DESTINATION($q, s$)
1    $k_1 \leftarrow A_q[q]$
2    $k_2 \leftarrow A_q[q+1]$
3    while $k_1 < k_2$ do
4        if $A_s[k_1] = s$ then
```

```
5           return $A[k_1]$
6         end
7         $k_1 \leftarrow k_1 + 1$
8     end
9     return fail
```

Lines 1 and 2 assign to k_1 and k_2 the indices in A where the transitions for state q begin and terminate, respectively. Lines 3-8 iterate over the transitions. Line 4 checks if the current transition is on the symbol s. If so, line 5 returns the destination state. If the iteration fails to find a transition on s, the algorithm returns a `fail` (which can be interpreted as a "dead state").

Table 2. Arcs to state ratios for surface to lexical transducers.

	A	B
Language	Arcs/State	$\|Q\| : \|S\|$
German	3.5	132343 : 244
French	3.8	94257 : 288
Mandarin	4.9	45813 : 17027

Let α represent the time it takes to access an array with a known index; the above algorithm requires $(3 + k_2 - k_1)\alpha$ if successful and $(2 + k_2 - k_1)\alpha$ otherwise. The worst case complexity takes place when requesting the destination on the last symbol of the alphabet from the state with the most arcs. However, considering that the ratio of arcs per state is small in natural language machines, especially lexical FSTs, $k_2 - k_1$ is always small. Table 2, Column A, gives empirical values from our inventory of surface-lexical transducers.

If a situation arises where the ratio of arcs per state is high (see §3.3 below), lines 3-8 in the above algorithm can be replaced with a more efficient search, e.g., binary search if the contents of A_s between k_1 and k_2 are sorted.

3.3 Row-Indexed Storage with Transpose

It was noted above that the space complexity above is $2|\delta| + |Q|$. In language and speech transducers, it is usually the case that $|S| \ll |Q|$ as shown in Table 2, Column B. Hence, further compression can be achieved by employing row-indexed storage on the transpose M^T of an original matrix M.

For machines where $|Q|$ is large there is a serious drawback in time complexity using this method in that $k_2 - k_1$ will be very large. In such a case, a binary search algorithm need to replace lines 3-8 of the algorithm as indicated above. Despite this drawback, this method may prove useful for special-purpose hardware devices where there is a real limitation in memory, but with a lot of "horse power."

3.4 Storing Weighted Finite-State Transducers

A weighted finite state transducer is a traditional transducer except that both arcs and final states are associated with costs (Mohri, Pereira, and Riley, 1998).

The same mechanism described above is used to compress transducers, albeit with some modification to cater for the symbols on the second tape. Here, transition are represented with a 2-dimensional matrix, i.e., a 3-dimensional array. The extra dimension is needed to store the symbols on the second tape of the transducer, i.e., the output symbols. The 2-dimensional matrix consists of four vectors: A, A_q and A_s are as above with an additional vector, A_o, of the same size as A_s to store output symbols.

In the case of weighted transducers, the weights need to be represented as well. An additional vector, A_w, is used to store weights on transitions. Its size is equal to the size of A_s.

3.5 Storing Final States

The representation of the machine needs to indicate which states are final. In the original Bell Labs implementation, final states have non-zero costs (final states with cost 0 are coded with a special constant `NO_COST` value). In the compressed version, final states are indicated by a bit vector whose size is $|S| \bmod 8 + 1$ octets. If a state i is final, then the ith bit in the array is set; otherwise, it is clear.

4 Further Compression Methods

Although the above method for storing sparse matrices results in good compression rates, much compression can be achieved by implementing a number of complementary ideas.

4.1 Using Every Bit

The underlying storage mechanism makes use of a data structure that stores an element using one, two, three or four bytes. The choice depends on the maximum value of a particular data type. The alphabet of our German FSTs is 244 symbols and can be coded using eight bits (one octet). The French system, on the other hand, has an alphabet of 288 symbols for which nine bits (two octets)are required.

One can get additional savings by pairing the input and output symbols. While this will not help in the case of German, tremendous savings can be achieved in the case of French: 16 bits (two octets), instead of 18 (three octets), will be sufficient to code pairs of symbols. Similar techniques are used throughout the implementation which yield almost double the compression rate.

4.2 Lossless Removal of Transitions

While the ratio of arcs to states for lexical-based FSTs is quite low (see Table 2, Column B), this is not the case in rule-based FSTs. Consider the [e]-deletion rule, depicted in its FST representation in Fig. 2.1. There are $|S| - 1$ arcs from state 0 to itself on all symbols of the alphabet apart from 'v'. If the alphabet size is that of the English alphabet (in addition to punctuation symbols), this FST would contain over 350 arcs.

Tremendous savings can be achieved by defining an 'other' symbol, denoted here by o, which was used in (Kay and Kaplan, 1983) and (Koskenniemi, 1983) as well as many current implementations. Hence, the $|S| - 1$ transitions from state 0 to itself will be replaced by one transition on $o{:}o$. Transitions such as those from state 6 to 0 pose a problem; here, the transition is on "other, but not g". This can be catered for by adding a "dead state" to the machine with a transition from state 6 to it on g:g. Then one can have a transition from state 6 to 0 on $o{:}o$. Care must be taken with the new machine, e.g., not to apply standard minimisation algorithms to it as the dead states will then be removed changing the expressiveness of the machine. An additional 10% of space saving can be achieved using this lossless removal of transitions (only 10% because not all FSTs are rule-based).

Even with the use of the 'other' symbol, further reduction is size can be achieved by algorithmically removing arcs from machines. Consider a text-to-phoneme that employs a number of FSTs in cascade. The FSTs at the beginning of the cascade are likely to deal with normalization issues. This input, as well as output, alphabets are probably the set of orthographic symbols. The FSTs at the other end of the cascade probably deal with only phonetic symbols and markers. However, rule compilers add backtracking arcs in all machines using the entire alphabet of the system (including orthographic, phonetic and other tagging symbols). Say the alphabet of the entire text-to-phoneme system makes use of markers such as `sing`, `pl`, these for sure will be used only in intermediate machines and can easily be removed from the FSTs at the font-end of the cascade.

The grammar writer can designate the set of real input symbols to the first FST in the cascade, say Σ_0. Performing the operation $Id(\Sigma_0) \circ FST_0$ (where Id is the identity operator and FST_0 is the first transducer in the cascade) will remove all unnecessary arcs in FST_0. Iterating over the arcs of the result, one computers the output alphabet of the new transducer, Σ_1. Performing the operation $Id(\Sigma_1) \circ FST_1$ will reduce the size of the next transducer, etc.

4.3 Extracting Weights

It was mentioned that weighted transducers associate a cost for each arc and for each final state. As costs are represented by a floating point number, this requires huge storage for large machines. One way to minimize this is by "pushing" the costs of all final states onto their incoming arcs. For example, say a final state is associated with a cost of 1.0 and has two incoming arcs with costs 2.0 and 3.0, respectively. The cost of the state can be added onto the costs of the incoming arcs. The result is freeing the final state from any cost value and assigning the costs of the incoming arcs to 3.0 and 4.0, respectively.

Additionally, it is usually the case that grammar writers use a limited number of unique weights in lexica and grammars. The entire German surface-lexical transducer employs 33 unique costs, and the French one employs a mere 12. These costs can be listed in an external vector and arcs will have indices into the vector. Instead of having a floating point value associated with each arc of the French FSTs, for example, 4 bits will suffice to index the 12 unique costs.

4.4 Storing More Information than Necessary!

Combining all of the above complementary methods saves a lot of space, but adds to the time complexity of accessing data. Our compression gives the user the option of storing in the machines information that is accessed heavily even though it can be computed at run time. For example, the composition algorithm (as well as others) ask of each state the numbers of transitions on ϵ as input and as output. This information can be computed at compile time and stored in the machine. Since the average of ϵ-containing transitions per state is low (input ϵ is 0.4 transitions per state on average), a few bits can be used for storing such information with a high gain in performance.

Caching mechanisms can also be used to keep the latest information in memory. Some of the machines that we implemented with caching perform faster than the original uncompressed machines.

5 Conclusion and Results

The above complementary methods were applied on the language modeling FSTs for multi-lingual TTS. The data from Table 1 is repeated in Table 3 with the compression results. For comparison purposes, the results of compressing the machines with Unix's `compress` are included (note that the machines cannot be accessed in this format).

Table 3. Storage requirement for letter-to-sound rules in the Bell Labs multilingual TTS system

Language	Modules	Size (MB)	Unix Compress	Our Compression (MB)
German	49	26.6	5.7	5.3
French	52	30.0	6.0	4.6
Mandarin	51	39.0	13.2	12.9

To illustrate access time, three German FSTs were composed with each other. Say that the composition on the original non-compressed machines takes 100% time. Applying the composition on the compressed versions of the machines (with 83% compression rate) requires 155% of the original time. If caching is enabled, however, the composition runs faster in 63% of the original time.

This paper outlined an eclectic method for compressing large natural language FSTs. Various complementary methods were employed yielding a high compression rate with minimal impact in speed.

Other related work include (Roche and Schabes, 1995) who mention using an algorithm by (Tarjan and Yao, 1979) to represent a sparse matrix. An anonymous reviewer kindly pointed out a work by (Liang, 1983) which I have no access to.

Acknowledgments. This work benefited from many discussions with Richard Sproat who contributed especially to the material in §§4.2 and 4.3. Thanks are due to Martin Jansche and Victor Faubert for useful comments.

References

Halle, M. and S. Keyser. 1971. *English Stress, Its Forms, Its Growth, and Its Role in Verse*. Studies in Language. Harper & Row, New York.

Kaplan, R. and M. Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–78.

Karttunen, L. and K. Beesley. 1992. Two-level rule compiler. Technical report, Palo Alto Research Center, Xerox Corporation.

Kay, M. and R. Kaplan. 1983. Word recognition. This paper was never published. The core ideas are published in Kaplan and Kay (1994).

Koskenniemi, K. 1983. *Two-Level Morphology*. Ph.D. thesis, University of Helsinki.

Liang, F. 1983. *Word Hy-phen-a-tion by Comp-uter*. Ph.D. thesis, Stanford Univeristy.

Mohri, M., F. Pereira, and M. Riley. 1998. A rational design for a weighted finite-state transducer library. In D. Wood and S. Yu, editors, *Automata Implementation*, Lecture Notes in Computer Science 1436. Springer, pages 144–58.

Mohri, M. and R. Sproat. 1996. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 231–8.

Ritchie, G., A. Black, G. Russell, and S. Pulman. 1992. *Computational Morphology: Practical Mechanisms for the English Lexicon*. MIT Press, Cambridge, MA.

Roche, E. and Y. Schabes. 1995. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–53.

Roche, E. and Y. Schabes, editors. 1997. *Finite-State Language Processing*. MIT Press.

Sproat, R., editor. 1997. *Multilingual Text-to-Speech Synthesis: The Bell Labs Approach*. Kluwer, Boston, MA.

Tarjan, R. and A. Yao. 1979. Storing a sparse table. *Communications of the ACM*, 22(11):606–11.

An Extendible Regular Expression Compiler for Finite-State Approaches in Natural Language Processing

Gertjan van Noord[1] and Dale Gerdemann[2]

[1] University of Groningen vannoord@let.rug.nl
[2] University of Tübingen dg@sfs.nphil.uni-tuebingen.de

Abstract. Finite-state techniques are widely used in various areas of Natural Language Processing (NLP). As Kaplan and Kay [12] have argued, regular expressions are the appropriate level of abstraction for thinking about finite-state languages and finite-state relations. More complex finite-state operations (such as contexted replacement) are defined on the basis of basic operations (such as Kleene closure, complementation, composition).
In order to be able to experiment with such complex finite-state operations the FSA Utilities (version 5) provides an *extendible* regular expression compiler. The paper discusses the regular expression operations provided by the compiler, and the possibilities to create new regular expression operators. The benefits of such an extendible regular expression compiler are illustrated with a number of examples taken from recent publications in the area of finite-state approaches to NLP.

1 Introduction

Finite-state techniques are widely used in various areas of Natural Language Processing (NLP). As Kaplan and Kay [12] have argued, regular expressions are the appropriate level of abstraction for thinking about finite-state languages and finite-state relations. More complex finite-state operations (such as contexted replacement) are defined on the basis of basic operations (such as Kleene closure, complementation, composition).

For instance, context sensitive rewrite rules have been widely used in several areas of natural language processing, including syntax, phonology and speech processing. Johnson [11] has shown that such rewrite rules are equivalent to finite state transducers under the assumption that they are not allowed to rewrite their own output. An algorithm for compilation into transducers was provided by Kaplan and Kay [12]. Improvements and extensions to this algorithm have been provided by Karttunen [13] [15] [14] and Mohri & Sproat [19]. Such algorithms take as their input regular expressions for the strings to be replaced and the left and right contexts, and produce a finite-state transducer. In other words, such an algorithm provides a new regular expression operator.

Many different variants of replacement operators have been proposed, depending on whether rewrite rules are interpreted left to right, right to left or in parallel; whether rewrite rules are required to use longest, shortest or all matches; whether rules are obligatory or optional; whether contexts should match the input side or the output side of the transductions etc. For this reason, it is crucial to be able to experiment with each of the various proposals in a flexible way.

Version 5 of the FSA Utilities [25] is an extended, rewritten and redesigned version of the FSA Utilities toolbox previously presented at the first WIA [24]. The FSA Utilities toolbox has been developed as a platform for experimenting with finite-state approaches in natural language processing. For this reason, the FSA Utilities toolbox is implemented in SICStus Prolog (cf. also section 5).

FSA5 provides a very flexible *extendible* regular expression compiler. Below, we present the basic regular expression operations provided by the compiler, and the possibilities to create new regular expression operators. We illustrate the exendible regular expression compiler with a number of examples taken from recent publications in the area of finite-state approaches to NLP.

2 Regular Expressions

Table 1 gives an overview of the basic regular expression operators provided by FSA5. Apart from the standard regular expression operators and extended regular expression operators for regular languages, the tool-box also provides regular expression operators for regular relations. For example, the expression

```
{a:b,b:c,c:a}*                                                    (1)
```

is the transducer which rewrites each `a` into a `b`, each `b` into a `c`, and each `c` into an `a`. Consider furthermore a transducer which removes each `b`, but which leaves each non-`b` in place:

```
{b:[],? -b}*                                                      (2)
```

In this example[1], the expression `? -b` is any symbol except `b`. An expression `Expr` denoting a regular language is automatically coerced in the context in which a transducer is expected into `identity(Expr)`. Here, `? -b` is automatically coerced into `identity(? -b)`, because it is unioned with a transducer. Composing the examples 1 and 2:

```
{a:b,b:c,c:a}* o {b:[],? -b}*                                     (3)
```

yields a transducer which removes each `a`, and transduces each `b` to a `c`, and each `c` to an `a`. For instance, the input `abcabcabc` yields `cacaca`.

In FSA5, such a regular expression could be turned into a transducer using the command:

```
% fsa -r '{a:b,b:c,c:a}* o {b:[],? -b}*' >  ex1.fa                (4)
```

[1] For technical reasons a space is required after each occurrence of the `?` meta-symbol.

Table 1. Basic regular expression operators in FSA5.

`[]`	empty string
`[E1,E2,...En]`	concatenation of `E1, E2 ...En`
`{}`	empty language
`{E1,E2,...En}`	union of `E1, E2 ...En`
`E*`	Kleene closure
`E^`	optionality
`~E`	complement
`E1-E2`	difference
`$ E`	containment
`E1 & E2`	intersection
`?`	any symbol
`A:B`	pair
`E1 x E2`	cross-product
`A o B`	composition
`domain(E)`	domain of a transduction
`range(E)`	range of a transduction
`identity(E)`	identity transduction
`inverse(E)`	inverse transduction

In this case, the resulting automaton is written to the file `ex1.fa` in FSA5 format. There are options to produce automata in many different formats, including formats for other finite-automata tool-boxes such as AT&T's `fsm` program [18] and various visualization formats (including `dot`, `vcg`, `daVinci`, LaTeX and `postscript`). Other interesting formats are as a Prolog or C program implementing the transduction.

FSA5 can also be used interactively. In that case a graphical user interface is provided from which regular expressions can be input. The resulting automata are then displayed on the screen, and the resulting automata can be tested with sample inputs. The availability of such a graphical user interface in combination with various visualization tools has enabled the use of FSA5 in teaching [3]. For more information on these and other possibilities refer to the FSA Home Page: `http://www.let.rug.nl/vannoord/Fsa/`. The FSA Home Page includes an on-line demo.

3 Extendible Regular Expression Operators

The regular expression compiler can be extended with new regular expression operators by providing one or more files defining these operators. The definitions are essentially of two types. In both cases, the actual definitions are written in (often very simple) Prolog. On the one hand, operators can be defined in terms of existing regular expression operators. On the other hand, regular expression operators can be defined by providing a direct implementation on the underlying automata. Many researchers prefer the first style. For instance, Kaplan & Kay [12] (p. 376) argue:

> The common data structures that our programs manipulate are clearly states, transitions, labels, and label pairs—the building blocks of finite automata and transducers. But many of our initial mistakes and failures arose from attempting also to think in terms of these objects. The automata required to implement even the simplest examples are large and involve considerable subtlety for their construction. To view them from the perspective of states and transitions is much like predicting weather patterns by studying the movements of atoms and molecules or inverting a matrix with a Turing machine. The only hope of success in this domain lies in developing an appropriate set of high-level algebraic operators for reasoning about languages and relations and for justifying a corresponding set of operators and automata for computation.

Paradoxically, Mohri & Sproat improve upon Kaplan & Kay's algorithm by taking precisely the opposite approach. Their algorithm is primarily presented in terms of manipulations upon states and transitions within automata. One could perhaps translate Mohri & Sproat's algorithm into a high-level calculus, but a great deal of efficiency would be lost in the process. It is a testimony to the flexibility of FSA5, that these two approaches can both be implemented and combined (cf. section 4.3).

New operators in terms of existing operators. A regular expression operator is defined as a pair `macro(ExprA,ExprB)` which indicates that the regular expression `ExprA` is to be interpreted as regular expression `ExprB`. For example, simple nullary regular expression operators (equivalent to abbreviatory devices found in tools such as `lex` and `flex`), can be defined as in the following example:

```
macro( vowel, {a,e,i,o,u} )                                    (5)
```

indicating that the operator `vowel/0` can be understood by assuming that every occurrence of `vowel` in a regular expression is textually replaced by `{a,e,i,o,u}`.

The same mechanism is used to define n-ary operators, exploiting Prolog variables. For instance, the containment operator `containment(Expr)` is the set of all strings which have as a sub-string any of the strings in `Expr`. This could be defined as follows:[2]

```
macro(containment(Expr), [? *,Expr,? *])                       (6)
```

Naturally, operators defined in this way can be part of the definition of other operators. For instance, the operator `free(A)` is the language of all strings which do not have any of the strings in `A` as a substring. This can be defined as:

```
macro(free(A), ~containment(A))                                (7)
```

We have found it useful to define boolean operators using this mechanism. In fact, if we use the universal language to stand for *true* and the empty language to

[2] Note that this operator is standardly available in FSA5. Many of the built-in operators in FSA5 are defined using the same technique.

stand for *false*, then the standard operators for intersection and union correspond to conjunction and disjunction:

```
macro(true,? *).                                                    (8)
macro(false,{}).
```

With these definitions we get the expected properties:

```
true & true   = true     {true,true}   = true                        (9)
true & false  = false    {true,false}  = true
false & true  = false    {false,true}  = true
false & false = false    {false,false} = false
```

The macros for true and false can also be used to define a conditional expression in the calculus. The operator `coerce_to_boolean` maps the empty language to the empty language, and any non-empty language to the universal language:

```
macro(coerce_to_boolean(E),                                         (10)
        range(E o (true x true))).

macro(if(Cond,Then,Else),
     {  coerce_to_boolean(Cond) o Then,
       ~coerce_to_boolean(Cond) o Else  }).
```

Various interesting properties of automata have been implemented which yield boolean values, such as the predicates *is_equivalent/2* for recognizers, and *is_functional/1* and *is_subsequential/1* for transducers (using the algorithms described in for instance [23]).

Regular expression operator definitions can also be recursive. The following example demonstrates furthermore that definitions can take the operands of the operator into account. The operator `set(List)` yields the union of the languages given by each of the expressions in the list `List`; `union(A,B)` is a built-in operator providing the union of the two languages `A` and `B`:

```
macro(set([]),'{}').                                                (11)
macro(set([H|T]),union(H,set(T))).
```

We can also exploit the fact that these definitions are directly interpreted in Prolog by providing Prolog constraints on such rules. This possibility is used in [7] to define a longest-match concatenation operator which implements the leftmost-longest capture semantics required by the POSIX standard (cf. section 4.4).

A simple example is a generalization of the operator `free`. Suppose we want to define an operator `free(N,Expr)` indicating the set of strings which do not contain more than `N` occurrences of `Expr`. This can be done as follows:

```
macro(free(N,X),~ [? *|List]) :-                                    (12)
    free_list(N,X,List).

free_list(0,X,[X,? *]).
free_list(N0,X,[X,? *|T]) :-
    N0 > 0, N is N0-1, free_list(N,X,T).
```

Another example is an implementation of the N-queens problem: how to place N queens on an N by N chess-board in such a way that no queen attacks any other queen. For any N we can create a regular expression generating exactly all strings of solutions. A solution to the N-queen problem is represented as a string of N integers between 1 and N. An integer *i* at position *j* in this string indicates that a queen is placed on the *i*-th column of the *j*-th row.

```
macro(n_queens(N), sigma(N)*                                        (13)
                 & length(N)
                 & columns(N)
                 & diagonals(N)
                 & reverse(diagonals(N)))
```

The operator `n_queens(N)` is defined as the intersection of a number of constraints. The first constraint, `sigma(N)*`, indicates that a solution must be a string of integers between 1 and `N`. The second constraint indicates that the length of the string must be `N`. The remaining constraints ensure that queens do not attack each other. The definition of `length` illustrates once more the use of Prolog to create a regular expression; the definition of `sigma/1` uses the `set` operator defined previously.

```
macro(length(N),List) :- length(List,N), fill_qm(List).            (14)

fill_qm([]).
fill_qm([? | T]) :- fill_qm(T).

macro(sigma(N),set(L)):-
    findall(C,between(1,N,C),L).

between(N,_,N).
between(N0,N,I) :-
    N1 is N0+1, N1 < N+1, between(N1,N,I).
```

The complete program is given in the appendix. For instance, the expression `n_queens(5)` produces the automaton in figure 1.

The mechanism described sofar to define new regular expression operators is already quite powerful. As another illustration, consider the problem of compiling a given finite automaton into a regular expression. This problem becomes trivial if we allow the introduction of new operators. Here is the definition of an operator 'fa/5' which describes an automaton as a listing of its components:

```
macro(fa(Sigma,States,Initials,Finals,Transitions),          (15)
  range(          % state-sym-state triples:
       ([[States,Sigma]*,States]
              & % no non-transition triples:
        free([States,Sigma,States]-Transitions)
              & % start in start-state:
        [Initials,? *]
              & % end in final state:
        [? *,Finals]
       )      o % get rid of state names:
       [[States x [],?]*,States x []]
     ))
```

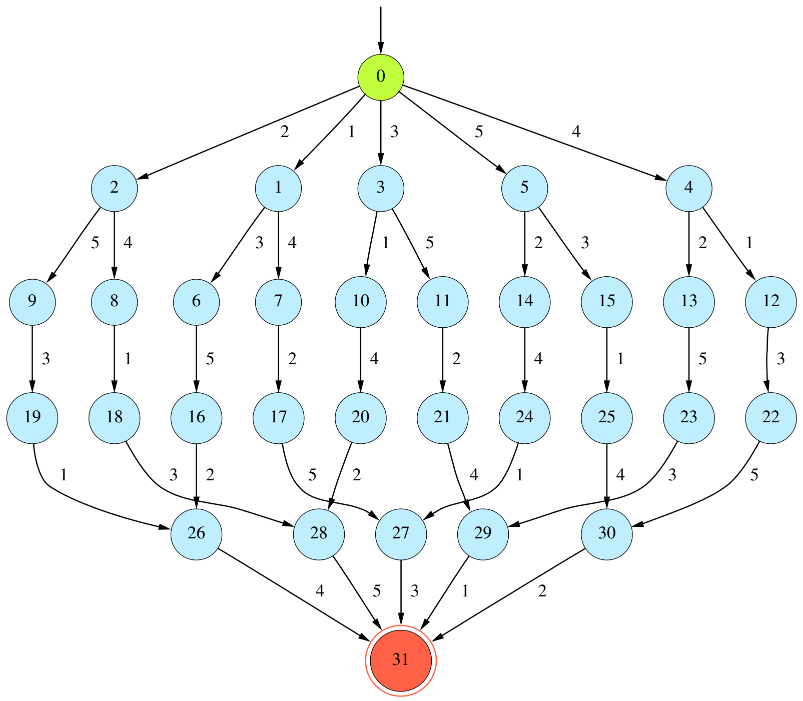


Fig. 1. Solution to the 5-queens problem

As an example, the automaton given in figure 2.16 of [10] (given in figure 2) can be specified as:

```
fa({0,1},{q1,q2,q3},{q1},{q2,q3},                            (16)
     {[q1,0,q2],[q1,1,q3],[q2,0,q1],
      [q2,1,q3],[q3,0,q2],[q3,1,q2]})
```

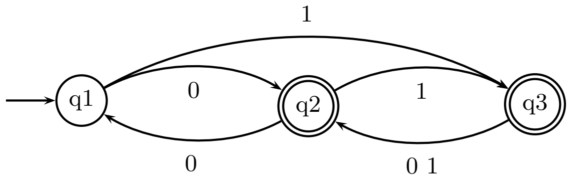


Fig. 2. Example automaton from figure 2.16 of [10].

Direct implementation of new operators. Some operators are more easily defined in terms of the underlying automaton. For instance, the operator `reverse(X)` is the set of all strings `Y` such that the reversal of `Y` is in `X`. In case the operand `X` is constructed by means of standard regular expression operators it would be possible to define the reverse operator recursively in terms of the various forms that `X` can take; in FSA however `X` could be constructed by means of various user-defined operators as well. Therefore this approach is not applicable. However, the operation is trivial to define in terms of the underlying automaton: each of the transitions needs to be swapped, final states become start states and vice versa. The (simplified) definition is given as follows:

```
rx(reverse(Expr),Fa) :-                                                    (17)
    fsa_regex:rx(Expr,Fa0), reverse_fa(Fa0,Fa).

reverse_fa(Fa0,Fa) :-
    fsa_data:start_states(Fa0,Finals),
    fsa_data:final_states(Fa0,Starts),
    fsa_data:transitions(Fa0,Trans0),
    reverse_transitions(Trans0,Trans),
    fsa_data:construct_fa(Starts,Finals,Trans,Fa).

reverse_trans([],[]).
reverse_trans([trans(A,B,C)|T0],[trans(C,B,A)|T]) :-
    reverse_trans(T0,T).
```

As is typical in such definitions, the `fsa_regex:rx` predicate is used to construct an automaton for a given regular expression. The `fsa_data` module provides a consistent interface to the internal representation of automata. Its predicates can be used to select relevant parts of an automaton (such as start states, final states and transitions) and to construct automata on the basis of such parts.

4 Regular Expression Operators in NLP

This section illustrates the flexibility and the power of the FSA5 extendible regular expression compiler on the basis of a number of examples taken from recent publications in the field of NLP.

4.1 Lenient Composition

In a recent paper, Karttunen [16] has provided a new formalization of Optimality Theory in terms of regular expressions. Optimality theory [20] is a framework for the description of phonological regularities which abandons rewrite rules. Instead, a universal function called GEN is proposed which maps input strings non-deterministically to many different output strings. In addition, a set of ranked universal constraints rule out many of the phonological representations generated by GEN. Some constraints can be conflicting. Therefore it might be impossible for a candidate string to satisfy all constraints. A string is allowed to violate a constraint as long as there is no other string which does not violate that constraint.

Procedurally, this mechanism can be understood as follows. Firstly, an input is mapped to a set of candidate output strings. This set of strings is then passed on to the most important constraint. This constraint removes many of the candidate strings. The remaining strings are passed on to the next important constraint, and so on. If the application of the constraint would remove all remaining candidate strings, then no strings are removed (constraints are *violable*). In the simplest case, only a single string survives all of the constraints. If none of the strings satisfy a given constraint, then the strings survive with the least number of violations of that constraint.

Karttunen formalizes GEN as a regular relation. Each of the constraints is itself a regular language allowing only the strings which satisfy the constraint (unless no strings satisfy the constraint). If the constraints were to be combined using ordinary composition, then the set of outputs would often be empty. Therefore, instead of composition Karttunen introduces an operation of *lenient_composition* which is closely related to a notion of defaults.

Informally, the *lenient_composition* of `S` and `C` is the composition of `S` and `C`, except for those elements in the domain of `S` that are not mapped to anything by `S o C`. Thus, it enforces the constraint `C` to those strings in `S` which have an output that satisfies the constraint:

```
macro(priority_union(Q,R), {Q, ~domain(Q) o R}).                (18)
macro(lenient_composition(S,C), priority_union(S o C,S)).
```

Here, `priority_union` of two transductions `Q` and `R` is defined as the union of `Q` and the composition of the complement of the domain of `Q` with `R`; i.e. we obtain all pairs from `Q`, and moreover for all elements not in the domain of `Q` we apply `R`. Lenient composition of `S` and `C` is defined as the priority union of the composition of `S` and `C` (on the one hand) and `S` (on the other hand); i.e. we obtain the composition of `S` and `C` and moreover for all inputs for which that composition is empty we retain `S`.

Consider the example

```
lenient_composition({b x [b,b],a x [b,b]*},[b,b,b]*)            (19)
```

The input transducer maps an `a` to an even number of `b`'s, and it maps a `b` to two `b`'s. If this transducer is leniently composed with the requirement that the result must be a string of `b`'s divisible by 3, then the resulting transducer maps b to two `b`'s, as before (since the constraint cannot be satisfied for any map of the input `b`), and it maps an `a` to a string of `b`'s which is divisible by 6.

Karttunen illustrates the method by providing a formalization of the syllabification analysis in Optimality Theory. This formalization has been implemented in FSA5 and is given in the appendix.

4.2 Priority Union for Lexical Analysis

Another application of the priority union operator is in spell checking. As in [4] we consider a finite-automaton approach. Suppose we are given a dictionary in the form of a transducer. The transducer will map each word to its lexicographic description. A spell checker attempts to find, for a given word, the lexicographic description of the word which is closest to a word in the dictionary according to some distance function. As in many spell checkers we assume Levenshtein distance: the minumum number of substitutions, deletions and insertions that is required to map a string into another. In FSA all strings with a Levenshtein distance of 1 can be defined as follows; here `X` can be thought of as the dictionary, `lev1(X)` is the Levenshtein-1 closure of the dictionary:

```
macro(lev1(X), { subs(X), del(X), ins(X) })
```
(20)

The operators `subs/1, del/1` and `ins/1` are built-in. The expression `subs(X)` stands for all pairs (x, y) such that (x', y) is in the relation defined by `X` and x' can be formed from x by a single substitution. The insertion and deletion operators are defined likewise.

In contrast to [4] we want to obtain the candidates with minimal distance. For instance, if we attempt to lookup `book` then we don't want to get the description of `cook` as a result. This can be defined using the priority union operator as follows:

```
macro(spell1(X), priority_union(X, lev1(X)))
```
(21)

For instance, applying `spell1` to a dictionary consisting of the identity transducer over the words *book, look, lock, oak* would map each of these words to itself, and in addition it would map a form such as *wook* to the set *book, look* and a form such as *ook* to the set *book, look, oak.*

We can define expressions for any given radius α. For example, the case which treats $\alpha = 2$ is given by:

```
macro(spell2(X), priority_union(spell1(X), lev1(lev1(X))))
```
(22)

4.3 The Replace Operator

In [19] a variant of the replace operator is implemented which is more efficient than previous implementations provided by Kaplan and Kay [12] and Karttunen [13]. This improved version crucially depends on the possibility of manipulating the transitions and states of the underlying automata directly. The replacement of expression `Phi` into `Psi` in the context of `Left` and `Right` is written `replace(Left,Phi,Psi,Right)`. In the left-to-right interpretation, this operator can be defined as the following cascade:

```
macro(replace(L,Phi,Psi,R),                                          (23)
    r(R) o f(Phi) o replace(Phi,Psi) o l1(L) o l2(L))
```

This definition and the definitions of the auxiliary operators are closely modelled on those given in [19]. The auxiliary operators are defined in the appendix.

A typical example of the use of the replace operator is provided by the past tense endings of Dutch regular verbs. In Dutch, the singular past tense is formed by the `-de` and `-te` suffixes. If the previous phoneme is voiced, the suffix `-de` must be used; in order circumstances the `-te` suffix is appropriate. This phenomenon can be analysed by assuming an underlying, abstract, `-Te` suffix. The `T` is then transformed into a `d` or `t` depending on context. The rule can be defined as follows (the `+` indicates a morpheme boundary):

```
macro(tkofschip,                                                     (24)
      replace([{k,f,s,[c,h],p,t,x},+],'T',t,e)
                            o
                   replace(+,'T',d, [])
```

4.4 Leftmost-Longest Contexted Replacement

In [7] a leftmost-longest match contexted replacement operator

`lml(T,Left,Right)`

is defined which ensures that the transducer `T` is applied in contexts `Left` and `Right`, using a leftmost-longest match strategy. One application of such an operator is finite-state parsing (chunking), [1,5,8,22]. In finite-state parsing, sets of context-free rules are collected into levels. Typically there is a finite number of such levels, and these levels are ordered. First each of the rules of the first level apply. The result is then input to the second level, etc. Note that rules cannot work on their own output, unless the same rule is placed in several levels.

In the following example we will not use the contexts; therefore lml/1 is defined as:

```
macro( lml(T), lml(T,[],[]) )                                        (25)
```

This operator ensures that the transducer T is applied to a string at all possible positions, using a left-to-right left-most longest match policy.

In this particular example we will assume that the input to the finite-state parser is a tagged sentence: each word is represented by a category, an opening bracket, the word itself, and a closing bracket. A rule with a given left hand side and right hand side will look for the sequence of elements described by the right hand side and wrap the result inside left hand side brackets. In general, the macro `-->` can be defined as the following transducer (this macro disallows the case that the daughters is the empty string):

```
macro((A --> Ds), [[] x [A,'['], Ds-[], []:']'])                    (26)
```

We use the macro `d(Expr)` for elements in the right hand side of rules; the macro `dw(Expr)` is similar but is used for pre-terminals, to refer to specific words.

```
macro(d(Cat),       [Cat, '[':'(',free(']'),']':')']).              (27)
macro(dw(Cat,Word),[Cat, '[':'(',   Word,  ']':')']).
```

Note that the brackets (introduced by an earlier level) are replaced here by other brackets in order to ensure that these brackets cannot be used in later levels again; in other words at any given level we can only 'see' the top-most constituents (yet, the full parse tree can be recoved using the 'invisible' brackets).

Using these two macro's a rule to recognize basic noun phrases is:

```
np --> [d(art)^,d(num)^,d(adj)*,d(n)+]                              (28)
```

A level of rules can now simply be defined as the replacement operator applied to the union of these rules. For instance, the following is a level recognizing multi-word-units (for instance, the Dutch phrase 'ten opzichte van' is comparable to the English phrase 'with respect to'):

```
macro(mwu,lml({                                                     (29)
  (p --> [dw(p,ten),dw(n,opzichte),dw(p,van)] ),
  (p --> [dw(p,in),dw(n,verband),dw(p,met)]   ),
  (p --> [dw(p,in),dw(n,plaats),dw(p,van)]    ) }))
```

Finally, we use composition to combine a number of such levels. Thus, the following expression defines a simple noun-phrase chunker:

```
macro(np_chunker,                                                   (30)
  mwu o lml(( adj --> [d(adv), d(adj)]))
      o lml((  np --> [d(art)^,d(num)^,d(adj)*,d(n)+]))
      o lml((  pp --> [d(p),d(np)]))
      o lml((  np --> [d(np),d(pp)+])))
```

For example, one of the sentences from the Eindhoven corpus [6] is chunked as in figure 3.

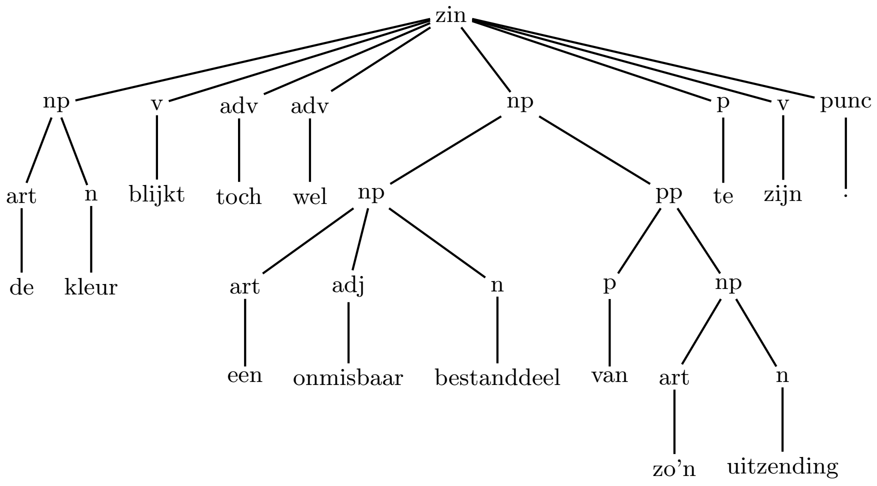


Fig. 3. Application of NP chunker

5 Implementational Issues

The regular expression compiler is defined in SICStus Prolog. This choice was motivated because the FSA Utilities toolbox has been developed as a platform for experimenting with finite-state approaches in natural language processing. Prolog allows for rapid prototyping of new techniques and variations of known techniques. The drawback is that the CPU-time requirements increase in comparison with an implementation based on C or C++. In [26] it is shown that the implementation of the determinizer is typically about 2 to 5 times slower in FSA Utilities than in AT&T's `fsm` library ([18]); the FSA Utilities toolbox contains a variant of the determinization algorithm for input automata with large amounts of ϵ-moves. In such cases FSA Utilities is often much faster. The implementation of the minimization algorithm [9,2] is up to three times faster than the implementation described in [17], which was shown to be much faster than the corresponding implementations in Fire Lite [27] and Grail [21].

Regular expressions are read and parsed using the Prolog parser (i.e. regular expressions are read in as Prolog terms), exploiting the inherent flexibility of this parser (such as the possibility to declare that new operators may be written using operator syntax; therefore we can write `E1 o E2` instead of `o(E1,E2)`). The constructed term is straightforwardly compiled into a corresponding finite automaton using a simple top-down recursive-descent procedure.

This mechanism implies that in order to construct an automaton for a regular expression such as `[a*,b,c^,d+]` automata are constructed for each of the sub-expressions. For regular expressions which are constructed solely using such simple operators, more efficient automaton construction algorithms are known. We have not implemented these algorithms because of the desire to be able to treat user-defined operators. A possible improvement could be to have the compiler identify which parts of an expression are simple enough to be treated by a more efficient specialized algorithm.

The compiler supports caching of sub-expressions. If the cache facility is switched on, then the result of each sub-expression that is encountered will be cached for later re-use. This can increase efficiency for the compilation of a single expression, but it is especially useful in an interactive session where the user gradually alters the regular expression; typically a large part of the expression remains the same and interactive response time can be much more attractive.

The caching facility can also be used selectively. The `cache(Expr)` operator can be used to cache the result of the compilation of a specific regular expression `Expr`. For instance, in example 20 the expression `lev1(X)` is defined as {`subs(X), del(X), ins(X)`}. If we write instead:

```
macro(lev1(X), {subs(cache(X)),del(cache(X)),ins(cache(X))}) (31)
```

then `X` will be compiled only once.

The compiler supports a number of other operators which have an effect on the underlying automata, but not on the corresponding language or relation. For instance, the operator `determinize(Expr)` can be used to ensure that the resulting automaton is determinized. Similar operators provide a simple interface to various minimization algorithms provided by FSA5.

Furthermore, certain operators can be used for the sole purpose of obtaining a side-effect. One example was the `cache/1` operator discussed above. The operator `spy(Expr)`, for instance, can be used to request that the compiler provides progress information on the compilation of the expression `Expr` (size of the result, and CPU-time required to obtain the result). Such progress information is crucial for a better understanding of the sources of complexity of particular expressions.

Concluding Remarks

We have presented the extendable regular expression compiler of FSA5. We have shown that the functionality and flexibility provided by the toolbox can be used to experiment with a variety of finite-state techniques in natural language processing, including applications in phonology, morphology and syntax.

References

[1] Steven Abney. Partial parsing via finite-state cascades. In John Carroll, editor, *Workshop on Robust Parsing; Eight European Summer School in Logic, Language and Information*, pages 8–15, 1995.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] Gosse Bouma. A modern computational linguistics course using dutch. In *EACL 99: Computer and Internet Supported Education in Language and Speech Technology. Proceedings of a Workshop sponsored by ELSNET and The Association for Computational Linguistics*, Bergen Norway, 1999.

[4] Christian S. Calude, Kai Salomaa, and Sheng Yu. Metric lexical analysis. In O. Boldt, H. Juergensen, and L. Robbins, editors, *Workshop on Implementing Automata; WIA99 Pre-Proceedings*, Potsdam Germany, 1999.

[5] Jean-Pierre Chanod and Pasi Tapanainen. A robust finite-state grammar for French. In John Carroll, editor, *Workshop on Robust Parsing*, Prague, 1996. These proceedings are also available as Cognitive Science Research Paper #435; School of Cognitive and Computing Sciences, University of Sussex.

[6] P. C. Uit den Boogaart. *Woordfrequenties in geschreven en gesproken Nederlands.* Oosthoek, Scheltema & Holkema, Utrecht, 1975. Werkgroep Frequentie-onderzoek van het Nederlands.

[7] Dale Gerdemann and Gertjan van Noord. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, Bergen Norway, 1999.

[8] Gregory Grefenstette. Light parsing as finite-state filtering. In *EACI 1996 Workshop Extended Finite-State Models of Language*, Budapest, 1996.

[9] John E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[10] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[11] C. Douglas Johnson. *Formal Aspects of Phonological Descriptions.* Mouton, The Hague, 1972.

[12] Ronald Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–379, 1994.

[13] Lauri Karttunen. The replace operator. In *33th Annual Meeting of the Association for Computational Linguistics*, M.I.T. Cambridge Mass., 1995.

[14] Lauri Karttunen. Directed replacement. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, 1996.

[15] Lauri Karttunen. The replace operator. In Emannual Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 117–147. Bradford, MIT Press, 1997.

[16] Lauri Karttunen. The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*, pages 1–12, Ankara, 1998.

[17] George Anton Kiraz and Edmund Grimley-Evans. Multi-tape automata for speech and language systems: A prolog implementation. In Derick Wood and Sheng Yu, editors, *Automata Implementation. Second Internation Workshop on Implementing Automata, WIA '97*, pages 87–103. Springer Lecture Notes in Computer Science 1436, 1998.

[18] Mehryar Mohri, Fernando C.N. Pereira, and Michael Riley. A rational design for a weighted finite-state transducer library. In *Automata Implementation. Second International Workshop on Implementing Automata, WIA '97.* Springer Verlag, 1998. Lecture Notes in Computer Science 1436.

[19] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, Santa Cruz, 1996.

[20] Alan Prince and Paul Smolensky. Optimality theory: Constraint interaction in generative grammar. Technical Report TR-2, Rutgers University Cognitive Science Center, New Brunswick, NJ, 1993. MIT Press, To Appear.

[21] D. Raymond and D. Wood. The grail papers. Technical Report TR-491, University of Western Ontario, Department of Computer Science, London Ontario, 1996.

[22] Emmanuel Roche. Parsing with finite-state transducers. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 241–281. MIT Press, Cambridge, 1997.

[23] Emmanuel Roche and Yves Schabes. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing.* MIT Press, Cambridge, Mass, 1997.

[24] Gertjan van Noord. FSA Utilities: A toolbox to manipulate finite-state automata. In Darrell Raymond, Derick Wood, and Sheng Yu, editors, *Automata Implementation*, pages 87–108. Springer Verlag, 1997. Lecture Notes in Computer Science 1260.

[25] Gertjan van Noord. FSA Utilities (version 5), 1998. The *FSA Utilities* toolbox is available free of charge under Gnu General Public License at http://www.let.rug.nl/~vannoord/Fsa/.

[26] Gertjan van Noord. The treatment of epsilon moves in subset construction. In *Finite-state Methods in Natural Language Processing*, Ankara, 1998. cmp-lg/9804003. Accepted for *Computational Linguistics.*

[27] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Eindhoven University of Technology, 1995.

A Syllabification in Optimality Theory

This is the implementation of Karttunen's formalization of syllabification in Optimality Theory.

```
%% Karttunen's X -> L ... R. Every X is ‘bracketed’ with L and R.
macro(dots(X,L,R), [[free(X), [[] x L, X, [] x R]]*, free(X)]).
%% Karttunen's A => L R. Every A must occur in context L _ R.
macro(restrict(A,L,R), ~[? *,A,~[R,? *]] & ~[~[? *,L],A,? *]).
macro(cons,{b,c,d,f,g,h,j,k,l,m,n,p,q,r,s,t,v,w,x,z}).
macro(lbr,{'O[', 'D[', 'X[', 'N['}).
macro(input,{cons,vowel}*).
macro(parse,   dots(cons,{'O[','D[','X['},']')
             o dots(vowel,{'N[','X['},']')).
macro(overparse,[([] x [lbr,']'])^,dots({cons,vowel},[],[lbr,']']^)]).
macro(onset,['O[', cons^, ']']).
macro(nucleus,['N[', vowel^, ']']).
macro(coda,['D[', cons^, ']']).
macro(unparsed,['X[', {cons,vowel}, ']']).
macro(syllable_structure,ignore([onset^,nucleus,coda^],unparsed)* ).
macro(gen, input o overparse o parse o syllable_structure).
macro(have_ons,restrict('N[', onset, [])).
macro(nocoda,free('D[')).
%% 'parse' is used twice in Karttunen 98; we use parsed(N) where N is
%% the maximum number of occurrences of X
macro(parsed(N), free(N,'X[')).
macro(fillnuc, free(['N[', ']'])).
macro(fillons, free(['O[', ']'])).
:- op(403,yfx,lc).
macro(R lc C, lenient_composition(R,C)).
macro(syllabify,gen lc have_ons lc nocoda lc fillnuc lc parsed(0) lc
        parsed(1) lc parsed(2) lc parsed(3) lc parsed(4) lc fillons ).
```

B Mohri & Sproat Replace Operator

Implementation in FSA5 of the contexted replacement operator of [19].

```
macro(r(R),reverse(marker(1,[sigma*,reverse(R)],[>]))).
macro(f(F),reverse(marker(1,[{sigma,>}*,reverse([ignore(F,{>}),>])],
                              ['<1','<2']))).
macro(l1(L), sloppy_ignore(marker(2,[sigma*,L],'<1'),{'<2':'<2'})).
macro(l2(L),marker(3,[sigma*,L],'<2')).
macro(replace(Phi,Psi), {{sigma,'<2':'<2', > :[]},
               ['<1':'<1',ignore(Phi,{'<1','<2',> }) x Psi,> :[]]}*).
macro(sigma,? - {'<1','<2',>}).

rx(marker(Type,Expr,C),Fa) :-
    fsa_regex:rx(identity(determinize(Expr)),Fa0), mark(Type,C,Fa0,Fa).

mark(1,Ins,Fa0,Fa) :-  %% Ins: symbols to be inserted
    fsa_regex:add_symbols(Ins,Fa0,Fa1), fsa_data:symbols(Fa1,Sig),
    fsa_data:start_states(Fa1,Starts),  fsa_data:transitions(Fa1,Trs0),
    fsa_data:final_states(Fa1,Fins),    fsa_data:all_states(Fa1,AllSts),
    ordsets:ord_subtract(AllSts,Fins,NFins0),
    add_ins(Fins,Ins,NFins,NFins0,Trs,Trs1),
    replace_trs_sf(Trs0,Trs1,Fa0),
    fsa_data:rename_fa(Sig,Starts,NFins,Trs,[],Fa).

replace_trs_sf([],[],_).
replace_trs_sf([trans(A0,B,C)|T0],[trans(A,B,C)|T],Fa):-
    ( fsa_data:final_state(Fa,A0) -> A=q(A0) ; A=A0 ),
    replace_trs_sf(T0,T,Fa).

add_ins([],_,F,F) --> [].
add_ins([F0|Fs],Ins,[q(F0)|NewF0],NewF) -->
    add_ins0(Ins,F0), add_ins(Fs,Ins,NewF0,NewF).

add_ins0([],_F) --> [].
add_ins0([Sym|Syms],F) --> [trans(F,[]/Sym,q(F))], add_ins0(Syms,F).

mark(2,Del,Fa0,Fa) :-  %% Sym is a symbol to be deleted
```

```
    fsa_regex:add_symbols([Del],Fa0,Fa1),
    fsa_data:copy_fa_except(transitions,Fa1,Fa2,Trs0,Trs),
    fsa_data:copy_fa_except(final_states,Fa2,Fa,Fins,AllSts),
    fsa_data:all_states(Fa0,AllSts),
    add_deletions(Fins,Del,Trs1,Trs0),  sort(Trs1,Trs).

add_deletions([],_) --> [].
add_deletions([F|Fs],Del) --> [trans(F,Del/[],F)], add_deletions(Fs,Del).

mark(3,Del,Fa0,Fa) :- %% Del is a symbol to be deleted
    fsa_regex:add_symbols([Del],Fa0,Fa1),
    fsa_data:copy_fa_except(transitions,Fa1,Fa2,Trs0,Trs),
    fsa_data:copy_fa_except(final_states,Fa2,Fa,Fins,AllSts),
    fsa_data:all_states(Fa0,AllSts),
    ordsets:ord_subtract(AllSts,Fins,NonFins),
    add_deletions(NonFins,Del,Trs1,Trs0),  sort(Trs1,Trs).

%% As defined by Mohri & Sproat. This should be done differently,
%% ignore is not defined for transducers.
macro(sloppy_ignore(A,B),ignore0(A,B)).
```

C N-Queens Problem

```
macro(free(Expr), ~containment(Expr)).
macro(sigma(N),set(L)):- findall(C,fsa_util:between(1,N,C),L).
macro(columns(N),Ints) :- columns(1,N,Ints).

%% don't use ordinary operator syntax, since this file is read-in with
%% regular expression operator precedences active.
columns(N,N,free([N,? *,N])).
columns(N0,N,free([N0,? *,N0]) & Ints) :-
    N0<N, is(N1,+(N0,1)), columns(N1,N,Ints).

macro(diagonals(N), I) :- diagonals(1,N,I).

diagonals(N0,N,I) :- is(N,N0+1),!, diagonals_n(1,N0,N,I).
diagonals(N0,N,I0 & I) :- diagonals_n(1,N0,N,I0),
    is(N1,+(N0,1)), diagonals(N1,N,I).

diagonals_n(N0,Br,N,I0) :- is(N,+(N0,Br)),!, diagonal(N0,Br,I0).
diagonals_n(N0,Br,N,I0 & I):-
    diagonal(N0,Br,I0), is(N1,+(N0,1)), diagonals_n(N1,Br,N,I).

diagonal(N0,Br,free([N0,length(MidN),N])) :-
    is(N,+(N0,Br)), is(MidN,-(Br,1)).
```

Multiset Processing by Means of Systems of Finite State Transducers[⋆]

Gheorghe Păun[1] and Gabriel Thierrin[2]

[1] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 70700 Bucureşti, Romania
gpaun@imar.ro

[2] Department of Computer Science, University of Western Ontario
N6A 5B7 London, Ontario, Canada
gab@csd.uwo.ca

Abstract. We introduce a computing mechanism of a biochemical inspiration (similar to a P system from the area of Computing with Membranes) which consists of a multiset of symbol-objects and a set of finite state transducers. The transducers process symbols in the current multiset in the usual manner. A computation starts in an initial configuration and ends in a halting configuration. The power of these mechanisms is investigated, as well as the closure properties of the obtained family. The main results say that (1) systems with two components and an unbounded number of states in each component generate all gsm images of all permutation closures of recursively enumerable languages, while (2) systems with two states in each component but an unbounded number of components can generate the permutation closures of all recursively enumerable languages, and (3) the obtained family is a full AFL. Result (2) is related to a possible (speculative) implementation of our systems in biochemical media.

1 Introduction

The present paper can be seen as a contribution both to Natural Computing, in the area of Computing with Membranes (P systems, see [8], [5], [10], [12], [13], [15], or the survey in [9]), and to Distributed (Parallel) Computing, in the multiset rewriting area (see, [1], [2], [3], etc.).

We start from the following speculation concerning a possible biochemical implementation of a finite state transducer acting on a multiset. Assume that the elements of the multiset are chemical compounds (for instance, DNA sequences) which swim in a given solution. A transition rule of the form $sa \rightarrow xs'$, where s, s' are states of the transducer, can be realized by a catalyst (for instance, an enzyme) which is able to change its state. That is, the catalyst C, in state s,

[⋆] Research supported by Grant OGP0007877 of the Natural Sciences and Engineering Research Council of Canada and, in the case of the first author, also by "Research for Future" Program no. JSPS-RFTF 96I00101, from the Japan Society for the Promotion of Science.

assists the compound a to evolve to the set of compounds x and changes the state to s' (thus, C is a "quasi-catalyst", having a "memory" of its actions; this is very much similar to P systems with bi-stable catalysts, introduced in [14], where catalysts with two possible states are used). Of course, if the number of states of a transducer is large, this idea has no chance to be implemented, so the natural question appears to consider only transducers with a small number of states. Transducers with a small number of states are weak, so the suggestion arises to put several such transducers to cooperate in a well defined manner.

In this way, we are led to the following kind of a "biochemical" computing mechanism. In a given space (a *membrane*) we have a multiset of objects, identified with symbols from a given alphabet. In the same space, we place several finite state transducers (generalized sequential machines). In a parallel manner, these transducers take symbols available around and, depending on their states, produce new symbols and change their states. In this way, a new *configuration* of the system is obtained. A sequence of such transitions among configurations is a *computation*; a computation is complete if it *halts*, that is, no further move is possible in its last configuration. In this way, a mapping from the initial multiset of objects to the multiset present in the halting configuration is defined. We can also associate a set of strings with a computation, as in P systems: we distinguish a *terminal* set of symbols and construct the string of terminal symbols appearing during the computation, in the order they are produced; when several terminal symbols are introduced at the same time, then any ordering of them is accepted (thus, several strings are associated with the same computation).

Consequently, we have here a variant of a P system, with only one membrane (so, a particular case from this point of view), but with the *evolution rules* of a powerful form: finite state machines, which remember by their states some information about their previous work. Still, such machines are among the simplest we can consider. (This could make appropriate the term *colony* for our device, in the sense introduced in [6], of a collectivity of as simple as possible devices working together.)

Somewhat expected (this happens in general in distributed systems, P systems included), the power of our computing machinery is rather large: systems with only two components are able to generate all recursively enumerable languages modulo a permutation; even the gsm images of permutation closures of recursively enumerable languages can be obtained in this way. If we left free the number of components, then systems with only two states in each component are sufficient in order to generate all permutation closures of recursively enumerable languages. This is a good result from the point of view of the possible (but never practically tried...) biochemical implementation: bi-stable catalysts suffice. Moreover, the family of languages generated by our devices is a full AFL (Abstract Family of Languages), so we can appreciate it as being very large.

2 Prerequisites

For elements of formal language theory we shall use below we refer to [16]. We only specify some notations.

For an alphabet V, V^* is the free monoid generated by V, λ is the empty string, and $V^+ = V^* - \{\lambda\}$. The length of $x \in V^*$ is denoted by $|x|$, while $|x|_a$ is the number of occurrences of the symbol a in the string x. The set of subwords of $x \in V^*$ is denoted by $Sub(x)$. If $V = \{a_1, \ldots, a_n\}$ (the ordering is important), then $\Psi_V(x) = (|x|_{a_1}, \ldots, |x|_{a_n})$ is the *Parikh vector* of the string $x \in V^*$. For a language $L \subseteq V^*$, $\Psi_V(L) = \{\Psi_V(x) \mid x \in L\}$ is the *Parikh set* of L and $p(L)$ is the permutation closure of L.

A Chomsky grammar is written in the form $G = (N, T, S, P)$, where N is the nonterminal alphabet, T is the terminal alphabet, S is the axiom, and P is the set of productions. We denote by RE the family of recursively enumerable languages.

In general, for a family FL of languages, we denote by pFL, FL^{one}, and FL^{bound} the families of permutation closures of languages in FL, of languages in FL over the one-letter alphabet, and of the strictly bounded languages in FL, respectively (a language $L \subseteq V^*$ is strictly bounded if there are n different symbols $a_1, \ldots, a_n \in V$ such that $L \subseteq a_1^* \ldots a_n^*$).

A notion which will be very useful below is that of a *matrix grammar*. Such a grammar is a construct $G = (N, T, S, M, C)$, where N, T are disjoint alphabets, $S \in N$, M is a finite set of sequences of the form $(A_1 \to x_1, \ldots, A_n \to x_n)$, $n \geq 1$, of context-free rules over $N \cup T$ (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases), and C is a set of occurrences of rules in M (N is the nonterminal alphabet, T is the terminal alphabet, S is the axiom, while the elements of M are called *matrices*).

For $w, z \in (N \cup T)^*$ we write $w \Longrightarrow z$ if there is a matrix $(A_1 \to x_1, \ldots, A_n \to x_n)$ in M and the strings $w_i \in (N \cup T)^*, 1 \leq i \leq n+1$, such that $w = w_1, z = w_{n+1}$, and, for all $1 \leq i \leq n$, either $w_i = w_i' A_i w_i'', w_{i+1} = w_i' x_i w_i''$, for some $w_i', w_i'' \in (N \cup T)^*$, or $w_i = w_{i+1}$, A_i does not appear in w_i, and the rule $A_i \to x_i$ appears in C. (The rules of a matrix are applied in order, possibly skipping the rules in C if they cannot be applied; we say that these rules are applied in the *appearance checking* mode.) If $C = \emptyset$, then the grammar is said to be without appearance checking (and C is no longer mentioned).

We denote by $\Longrightarrow^*$ the reflexive and transitive closure of the relation $\Longrightarrow$. The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac}. When we use only grammars without appearance checking, then the obtained family is denoted by MAT.

A matrix grammar $G = (N, T, S, M, C)$ is said to be in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \dagger\}$, with these three sets mutually disjoint, and the matrices in M are of one of the following forms:

1. $(S \to XA)$, with $X \in N_1, A \in N_2$,
2. $(X \to Y, A \to x)$, with $X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*$,

3. $(X \to Y, A \to \dagger)$, with $X, Y \in N_1, A \in N_2$,
4. $(X \to \lambda, A \to x)$, with $X \in N_1, A \in N_2$, and $x \in T^*$.

Moreover, there is only one matrix of type 1 and C consists exactly of all rules $A \to \dagger$ appearing in matrices of type 3. One sees that $\dagger$ is a trap-symbol; once introduced, it is never removed. A matrix of type 4 is used only once, at the last step of a derivation.

According to Lemma 1.3.7 in [4], for each matrix grammar there is an equivalent matrix grammar in the binary normal form.

A *multiset* over an alphabet $V = \{a_1, \ldots, a_n\}$ is a mapping $\mu : V \longrightarrow \mathbf{N} \cup \{\infty\}$. A multiset can be given in the form $\{(a_1, \mu(a_1)), \ldots, (a_n, \mu(a_n))\}$ or can be represented by any string $w \in V^*$ such that $\Psi_V(w) = (\mu(a_1), \ldots, \mu(a_n))$. We shall make below an extensive use of the string representation of a multiset. For the sake of mathematical accuracy, the multiset $\{(a_1, |w|_{a_1}), \ldots, (a_n, |w|_{a_n})\}$ represented by a string $w \in V^*$ is denoted by $\mu(w)$.

For $a_i \in V$ and a multiset μ over V, we say that a_i belongs to μ and we write $a_i \in \mu$ if $\mu(a_i) \geq 1$.

We say that the multiset μ_1 is included in the multiset μ_2, and write $\mu_1 \subseteq \mu_2$, if $\mu_1(a) \leq \mu_2(a)$ for all $a \in V$. The union of μ_1, μ_2 is the multiset defined by $(\mu_1 \cup \mu_2)(a) = \mu_1(a) + \mu_2(a)$, for all $a \in V$. The difference of two multisets, $\mu_1 - \mu_2$, is defined here only when $\mu_2 \subseteq \mu_1$, by $(\mu_1 - \mu_2)(a) = \mu_1(a) - \mu_2(a)$, for all $a \in V$.

3 P Systems of Transducers

We now introduce the computing mechanisms we investigate in this paper.

Let us first recall that a *gsm* (generalized sequential machine) is a construct $\gamma = (K, V_1, V_2, s_0, F, P)$, where K, V_1, V_2 are alphabets (the set of states, the input and output alphabets, respectively), $s_0 \in K$ (initial state), $F \subseteq K$ (final states), and P is a finite set of rewriting rules of the form $sa \to xs'$, for $s, s' \in K$ and $a \in V_1, x \in V_2^*$.

For $s, s' \in K, y_1, x \in V_2^*, y_2 \in V_1^*, a \in V_1$ we write $y_1 s a y_2 \Longrightarrow y_1 x s' y_2$ if $sa \to xs' \in P$. Then, for $w \in V_1^*$ we define $\gamma(w) = \{z \in V_2^* \mid s_0 w \Longrightarrow^* z s_f$, for some $s_f \in F\}$. For $L \subseteq V_1^*$, we define $\gamma(L) = \bigcup_{x \in L} \gamma(w)$. We say that $\gamma(L)$ is the image of L by the gsm γ.

For a family FL of languages, we denote by $gsm(FL)$ the family of gsm images of languages in FL.

Here we consider the gsm's not as operating on strings (and languages), but as *operators* on multisets of symbols; in this case, the final states are no longer necessary.

A *system of transducers* (in order to remind the Membrane Computing, we say, shortly, a *PT system*) of degree $n, n \geq 1$, is a construct

$$\Pi = (V, T, w_0, \gamma_1, \ldots, \gamma_n),$$

where:

- V is an alphabet (its elements are called *objects*),
- $T \subseteq V$ (the *terminal* alphabet),
- $w_0 \in (V - T)^*$ represents a multiset over V (the *initial* multiset),
- $\gamma_i = (K_i, V, s_{0,i}, P_i)$, $1 \le i \le n$, where K_i is a finite alphabet (set of states), $s_{0,i} \in K_i$, and P_i is a finite set of transition rules of the form $sa \rightarrow xs'$, for $a \in V - T, x \in V^*, s, s' \in K_i$ (thus, each γ_i is a gsm without final states and with identical input and output alphabets, namely equal to V; we say that γ_i is a *component* of Π).

Note that w_0 is a string over $V - T$ representing a multiset and that the terminal symbols cannot be processed by the rules in the components of Π.

Any $(n+1)$-tuple $(w, s_1, \ldots, s_n)$, with $w \in V^*, s_i \in K_i, 1 \le i \le n$, is called a *configuration* of Π; $(w_0, s_{0,1}, \ldots, s_{0,n})$ is the *initial configuration* of Π.

For two configurations $(w, s_1, \ldots, s_n)$, $(w', s'_1, \ldots, s'_n)$ we write $(w, s_1, \ldots, s_n) \Longrightarrow (w', s'_1, \ldots, s'_n)$ (and we say that we have a *transition* between the two configurations) if the following conditions hold:

1. there is $k \ge 1$ and there are the indices $i_1, \ldots, i_k \in \{1, 2, \ldots, n\}$ such that
   - $\mu(a_{i_1} \ldots a_{i_k}) \subseteq \mu(w)$, for $a_{i_j} \in V, 1 \le j \le k$,
   - $s_{i_j} a_{i_j} \rightarrow x_{i_j} s'_{i_j} \in P_{i_j}$, for $1 \le j \le k$,
   - $\mu(w') = (\mu(w) - \mu(a_{i_1} \ldots a_{i_n})) \cup (\mu(x_{i_1}) \cup \mu(x_{i_2}) \cup \ldots \cup \mu(x_{i_k}))$,
   - for $l \in \{1, 2, \ldots, n\} - \{i_1, \ldots, i_k\}$ we have $s_l = s'_l$;
2. the set $\{i_1, \ldots, i_k\}$ is maximal, in the sense that there is no transition $s_r a_r \rightarrow x_r s'_r \in P_r$ for some $r \in \{1, 2, \ldots, n\} - \{i_1, \ldots, i_k\}$, such that $a_r \in \mu(w) - \mu(a_{i_1} \ldots a_{i_k})$ (no further object in the multiset $\mu(w)$ can be processed by a gsm different from those mentioned at the previous point, $\gamma_{i_1}, \ldots, \gamma_{i_k}$).

In plain words, each gsm which can use a transition rule must do it; if this is not possible, then we remain in the same state.

A configuration $(w, s_1, \ldots, s_n)$ is said to be a *halting* one if there is no configuration $(w', s'_1, \ldots, s'_n)$ such that a transition $(w, s_1, \ldots, s_n) \Longrightarrow (w', s'_1, \ldots, s'_n)$ is possible.

As usual, we denote by $\Longrightarrow^*$ the reflexive and transitive closure of the relation $\Longrightarrow$. A sequence of transitions is called a (complete) *computation* if it starts in the initial configuration and ends in a halting configuration.

There are several possibilities of associating a *result* with a computation. We choose the variant also followed in the P systems area, see [8], [9]: we collect the terminal symbols, in the order in which they are introduced, and form a string; if several terminal symbols are introduced at the same time (by the same component of Π, using a rule $sa \rightarrow xs'$ with x being a string, or by several components), then all the orderings of those symbols are allowed, hence a set of strings is associated with the same computation. The set of strings of this form is the language *generated* by Π and it is denoted by $L(\Pi)$.

Formally, this language is defined as follows. For two strings $w, w' \in V^*$ such that $\Psi_T(w) \le \Psi_T(w')$ (componentwise), we denote by $L(w' - w)$ the set of words $x \in T^*$ such that $\Psi_T(w') = \Psi_T(w) + \Psi_T(x)$ (the set of strings over T composed of symbols which appear in w' and not in w). Then, for a halting computation $\Delta : (w_0, s_{0,1}, \dots, s_{0,n}) \Longrightarrow (w_1, s_{1,1}, \dots, s_{1,n}) \Longrightarrow \dots \Longrightarrow (w_m, s_{m,1}, \dots, s_{m,n})$, we define

$$L(\Delta) = L(w_1 - w_0)L(w_2 - w_1)\dots L(w_m - w_{m-1}).$$

The language $L(\Pi)$ is the union of all languages $L(\Delta)$, for Δ being a halting computation with respect to Π.

We denote by PTL_n the family of languages generated by PT systems of degree less than or equal to $n, n \geq 1$; the union of all these families is denoted by PTL.

Directly from the definitions, we get:

Lemma 1. $PTL_n \subseteq PTL_{n+1},\ n \geq 1.$

4 An Example

In order to illustrate the definition and the work of a PT system, let us consider the system (of degree 3)

$$\begin{aligned}
&\Pi = (V, T, w_0, \gamma_1, \gamma_2, \gamma_3),\ \text{with}\\
&V = \{a, a', a'', \bar{a}, b, c, d, e, f, g, h\},\\
&T = \{a\},\\
&w_0 = a'a'b^n d,\ \text{for some } n \geq 1,
\end{aligned}$$

and the following components:

$$\begin{aligned}
\gamma_1 &= (\{s_{0,1}, s_{1,1}\}, V, s_{0,1}, P_1),\\
P_1 &= \{s_{0,1}b \to cs_{1,1},\ s_{1,1}h \to s_{0,1}\},\\
\gamma_2 &= (\{s_{0,2}, s_{1,2}, s_{2,2}, s_{3,2}, s_{4,2}\}, V, s_{0,2}, P_2),\\
P_2 &= \{s_{0,2}a' \to as_{4,2},\ s_{4,2}a' \to as_{4,2},\ s_{4,2}b \to bs_{4,2},\ s_{4,2}c \to cs_{4,2},\\
&\quad s_{0,2}d \to ds_{0,2},\ s_{0,2}c \to s_{1,2},\ s_{1,2}a' \to a''a''s_{1,2},\ s_{1,2}e \to fs_{2,2},\\
&\quad s_{2,2}g \to s_{3,2},\ s_{3,2}a'' \to a's_{3,2},\ s_{3,2}e \to fs_{0,2}\},\\
\gamma_3 &= (\{s_{0,3}, s_{1,3}, s_{2,3}, s_{3,3}, s_{4,3}\}, V, s_{0,3}, P_3),\\
P_3 &= \{s_{0,3}d \to ds_{0,3},\ s_{0,3}d \to eds_{1,3},\ s_{1,3}a' \to \bar{a}s_{2,3},\ s_{2,3}\bar{a} \to \bar{a}s_{2,3},\\
&\quad s_{1,3}f \to gs_{3,3},\ s_{3,3}d \to ds_{3,3},\ s_{3,3}d \to eds_{4,3},\ s_{4,3}a'' \to \bar{a}s_{2,3},\\
&\quad s_{4,3}f \to hs_{0,3}\}.
\end{aligned}$$

The components of this PT system are represented graphically in Figure 1.

The system works as follows (and halts in a configuration which contains 2^n copies of the symbol a).

If in state $s_{0,2}$ the component γ_2 chooses to go to state $s_{4,2}$, then we never come back to $s_{0,2}$. Assume that at the first step γ_2 uses the rule $s_{0,2}d \to ds_{0,2}$. Simultaneously, γ_1 transforms one occurrence of b in c and γ_3 remains in the initial state (the occurrence of d is used by γ_2). At the next step, γ_2 can pass to

$s_{1,2}$, by the rule $s_{0,1}c \to s_{1,2}$, making use of the symbol c introduced by the first component; γ_1 will wait in state $s_{1,1}$ until a symbol h is produced.

In state $s_{1,2}$, we can replace each a' by two copies of a''. Assume that at this time γ_3 remains in state $s_{0,3}$. The component γ_2 cannot leave $s_{1,2}$ before having produced a copy of e in γ_3. At any moment, γ_3 can use the rule $s_{0,3}d \to eds_{1,3}$. If in the current multiset we still have occurrences of a', then γ_3 will now introduce $\bar{a}$, passing to state $s_{2,3}$ (this is obligatory, because the symbol f is not available). The computation will never stop, because of the rule $s_{2,3}\bar{a} \to \bar{a}s_{2,3}$ which can be used forever. No output is obtained in such a case. Therefore, the rule $s_{0,3}d \to eds_{1,3}$ in γ_3 should be used after transforming all symbols a' into a'' (doubling them).

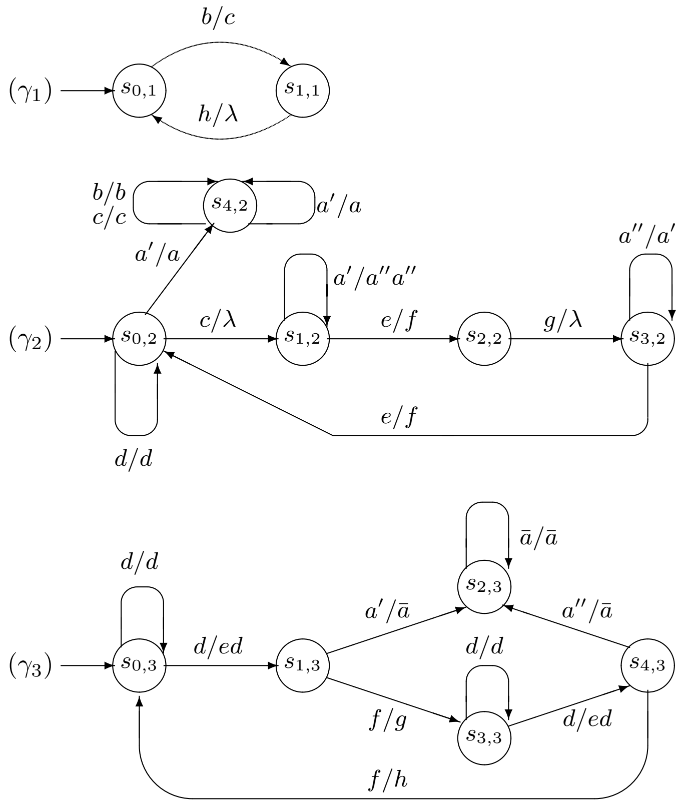


Fig. 1. Example of a PT system.

After using $s_{1,2}e \to fs_{2,2}$ in γ_2, we can use $s_{1,3}f \to gs_{3,3}$ in γ_3. At the next step, γ_2 can use g in order to pass to state $s_{3,2}$ while γ_3 stays in $s_{3,3}$ any number of steps. During this time, γ_2 replaces each a'' by a' (in state $s_{3,2}$). Again we can control whether or not this process is complete, by means of symbols $d, e, \bar{a}$: if γ_3 uses the rule $s_{3,3}d \to eds_{4,3}$ while symbols a'' are still present, then at the next step γ_3 will introduce the trap-symbol $\bar{a}$ (we cannot use the rule $s_{4,3}f \to hs_{0,3}$, because no symbol f is available).

If all symbols a'' were replaced by a', then γ_3 introduces the symbol h and returns to its initial state; note that γ_2 is already in the initial state. Using the symbol h, also γ_1 can return to its initial state. In the current multiset, the number of occurrences of a' is doubled in comparison with the number of such symbols in the previous configuration. During this time, a copy of b has been transformed in c.

When γ_2 enters the state $s_{4,2}$, we have two possibilities. If any copy of b or of c is present, then the computation will continue forever. If no copy of b or of c is present, then the computation can continue only until all symbols a' are replaced by a and then it stops: the component γ_3 can proceed further only using occurrences of the symbol f and such symbols are produced only by γ_2, which is no longer able to introduce f.

In conclusion, we can double n times the number of occurrences of a, that is, we stop in a configuration which contains 2^n copies of a. We may say that the system Π above computes the function $f(n) = 2^n, n \geq 1$.

One can modify this system in order to generate the language $\{a^{2^n} \mid n \geq 1\}$, but we leave this task to the reader.

5 The Power of PT Systems

We pass now to investigating the generative power of PT systems. The main result in this sense is the next one, showing that our mechanisms are very powerful.

Theorem 2. *$gsm(pRE) \subset PTL_2$, strict inclusion.*

Proof. (1) Let us first prove the inclusion.

It is known that $RE = MAT_{ac}$; this implies that $pRE = pMAT_{ac}$. Consider a matrix grammar with appearance checking $G = (N, T, S, M, C)$ and a gsm $\gamma = (K, T, V_2, q_0, F, P)$. Assume that G is in the binary normal form (hence $N = N_1 \cup N_2 \cup \{S, \dagger\}$) and that it contains k matrices of the form $m_i : (X_i \to \alpha_i, A_i \to x_i)$, $1 \leq i \leq k$, for some $X_i \in N_1, \alpha_i \in N_1 \cup \{\lambda\}$, $A_i \in N_2, x \in (N_2 \cup T)^*$, and n matrices of the form $m_j : (X_j \to Y_j, A_j \to \dagger)$, $k+1 \leq j \leq k+n$, for some $X_j, Y_j \in N_1, A_j \in N_2$.

For a string $x \in (N_2 \cup T)^*$ we denote by $\bar{x}$ the string obtained by replacing each terminal symbol a which appears in x by $\bar{a}$ (the nonterminal symbols remain unchanged).

We construct the PT system (of degree 2)

$$\begin{aligned} \Pi &= (V, V_2, w_0, \gamma_1, \gamma_2), \\ V &= N_1 \cup N_2 \cup V_2 \cup \{c, d, e, \dagger\} \cup \{\bar{a} \mid a \in T\}, \\ w_0 &= XAc, \text{ for } (S \to XA) \in M, X \in N_1, A \in N_2, \end{aligned}$$

with the following components:

$$\begin{aligned}
\gamma_1 &= (K_1, V, s_{0,1}, P_1),\\
K_1 &= K \cup \{q_f\} \cup \{s_{i,1} \mid 0 \leq i \leq k+n\}\\
&\quad \cup \{[q,r,i] \mid r : qa \to zq' \in P, q, q' \in K, a \in T, z \in V_2^*, 1 \leq i \leq |z|, |z| \geq 2\},\\
P_1 &= \{s_{0,1}X_i \to s_{i,1},\ s_{i,1}A_i \to \alpha_i\bar{x}_i s_{0,1},\ s_{i,1}c \to \dagger s_{i,1} \mid 1 \leq i \leq k\}\\
&\quad \cup \{s_{0,1}X_j \to e s_{j,1},\ s_{j,1}d \to Y_j s_{0,1},\ s_{j,1}A_j \to \dagger s_{j,1} \mid k+1 \leq j \leq k+n\}\\
&\quad \cup \{s_{0,1}c \to cq_0\}\\
&\quad \cup \{qc \to cq \mid q \in K\}\\
&\quad \cup \{q\bar{a} \to \alpha q' \mid qa \to \alpha q' \in P, \alpha \in V_2 \cup \{\lambda\}\}\\
&\quad \cup \{q\bar{a} \to a_1[q,r,1],\ [q,r,1]c \to ca_2[q,r,2], \dots,\\
&\qquad [q,r,t-2]c \to ca_{t-1}[q,r,t-1],\ [q,r,t-1]c \to ca_t q' \mid\\
&\qquad \text{for } r : qa \to zq' \in P, z = a_1a_2\dots a_t, t \geq 2, a_i \in V_2, 1 \leq i \leq t\}\\
&\quad \cup \{qc \to cq_f \mid q \in F\}\\
&\quad \cup \{q_f\alpha \to \alpha q_f \mid \alpha \in N_1 \cup N_2 \cup \{\bar{a} \mid a \in T\} \cup \{\dagger\}\},\\
\gamma_2 &= (\{s_{0,2}\}, V, s_{0,2}, P_2),\\
P_2 &= \{s_{0,2}e \to ds_{0,2}\} \cup \{s_{0,2}\alpha \to \alpha s_{0,2} \mid \alpha \in N_1 \cup N_2 \cup \{\dagger\}\}.
\end{aligned}$$

Let us examine the work of this system.

We start from the multiset represented by XAc; as long as a nonterminal symbol of G is present, the component γ_2 cannot stop. Therefore, we can halt only when no nonterminal symbol is present.

If γ_1 moves from $s_{0,1}$ to q_0 (the initial state of the gsm γ), then it never returns to $s_{0,1}$. From q_0, we simulate the work of γ, in the following way.

First, note that in each state $q \in K$ we can work forever using the rule $qc \to cq$. Thus, we have to reach the state q_f. Also in this state we can work forever if a nonterminal symbol of G is present or any symbol of the form $\bar{a}$, for $a \in T$, is present. Such barred symbols are introduced by the rules which simulate matrices in M (see below). Consequently, after entering the state q_0, we can finish the work of γ_1 only if no nonterminal is present and all terminals which are present (in the barred form) are correctly parsed by the gsm γ.

The parsing through γ can be simulated at any time; in particular, we can do that after eliminating all the nonterminals (this means that a derivation in G is completely simulated, see below). This is ensured by the fact that in each state $q \in K$ we can wait as much as we need, by using the rule $qc \to cq$. In this way, we have at our disposal all the terminal symbols, hence we can process them in any order we want. Otherwise stated, any permutation of a string generated by G is available and we can translate it. Note also the important fact that the rules of the form $q\bar{a} \to a_1[q,r,1]$, $[q,r,1]c \to ca_2[q,r,2], \dots, [q,r,t-1]c \to ca_tq'$, corresponding to a rule $r : qa \to a_1 \dots a_tq'$ from P, introduce the symbols $a_1, \dots, a_t$ one by one, hence in the order imposed by the rule r (if all these symbols were produced at the same time, then any permutation of them has to be considered, which is not correct).

What remains is to show that each derivation with respect to G can be simulated in Π.

Assume that we are in a configuration $(w, s_{0,1}, s_{0,2})$ (initially, $w = XAc$).

If we use a rule $s_{0,1}X_i \to s_{i,1}$ in γ_1, for $m_i : (X_i \to \alpha_i, A_i \to x_i)$ in M, then at the next step we have to use the rule $s_{i,1}A_i \to \alpha_i \bar{x}_i s_{0,1}$. Indeed, if we introduce the symbol †, by using the rule $s_{i,1}c \to \dagger s_{i,1}$, then the computation never stops. This means that the use of the matrix m_i is correctly simulated, both its rules were used. The process can be iterated.

If in the configuration $(w, s_{0,1}, s_{0,2})$ we use a rule $s_{0,1}X_j \to es_{j,1}$ for some matrix $m_j : (X_j \to Y_j, A_j \to \dagger), k+1 \leq j \leq k+n$, from M, then in state $s_{j,1}$ we have two possibilities.

If the symbol A_j is present in the current configuration, then we have to use the rule $s_{j,1}A_j \to \dagger s_{j,1}$ and the computation will never finish. If the symbol A_j is not present, then γ_1 cannot work, we remain in state $s_{j,1}$ until the component γ_2 uses the rule $s_{0,2}e \to ds_{0,2}$ (the symbol e is now available). Using the symbol d introduced by γ_2, γ_1 returns to its initial state and the symbol Y_j is introduced. One can see that again we simulate correctly a matrix in M, namely one with a rule used in the appearance checking manner. The process can be iterated.

In all cases, we get computations which can halt only when we correctly simulate the matrices of G. As we have seen above, when the derivation in G which is simulated by Π is terminal, and only in this case, we can also terminate the computation, reaching a halting configuration. In conclusion, $L(\Pi)$ consists exactly of all strings $w \in \gamma(z)$, for z being a permutation of a string in $L(G)$. Therefore, $L(\Pi) = \gamma(p(L(G)))$, hence $gsm(pRE) \subseteq PTL_2$.

(2) This is a proper inclusion. Indeed, let us consider the language

$$L = \{aba^2ba^4b \ldots ba^{2^n}ba^{2^{n+1}}b \ldots ba^{2^m} \mid m \geq 1\}.$$

This language can be generated by the matrix grammar $G = (\{S, A, A', X, Y, Z, \dagger\}, \{a, b\}, S, M, C)$, with the following matrices:

0. $(S \to XA)$,
1. $(X \to X, A \to aA'A')$,
2. $(X \to bY, A \to \dagger)$,
3. $(Y \to Y, A' \to A)$,
4. $(Y \to X, A' \to \dagger)$,
5. $(Y \to bZ, A' \to \dagger)$,
6. $(Z \to Z, A \to a)$,
7. $(Z \to \lambda)$.

One can see that the blocks a^{2^n} (by a "block" we understand a maximal subword consisting of occurrences of a) from the strings of L are produced in sequence, from left to right.

As in the first part of the proof, we can construct a PT system which simulates the work of G; because we use no gsm, the construction should be slightly modified. We give only the transitions of the two transducers, the states and the alphabets are obvious:

$$\begin{aligned} P_1 = \{ & s_{0,1}X \to s_{1,1},\ s_{1,1}A \to XaA'A's_{0,1},\ s_{1,1}c \to \dagger s_{1,1}, \\ & s_{0,1}X \to es_{2,1},\ s_{2,1}A \to \dagger s_{2,1},\ s_{2,1}d \to bYs_{0,1}, \\ & s_{0,1}Y \to s_{3,1},\ s_{3,1}A' \to YAs_{0,1},\ s_{3,1}c \to \dagger s_{3,1}, \end{aligned}$$

$$\begin{aligned}
&\quad s_{0,1}Y \to es_{4,1},\ s_{4,1}A' \to \dagger s_{4,1},\ s_{4,1}d \to Xs_{0,1},\\
&\quad s_{0,1}Y \to es_{5,1},\ s_{5,1}A' \to \dagger s_{5,1},\ s_{5,1}d \to bZs_{0,1},\\
&\quad s_{0,1}Z \to s_{6,1},\ s_{6,1}A \to as_{6,1}\},\\
P_2 &= \{s_{0,2}e \to ds_{0,2}\}\\
&\quad \cup \{s_{0,2}\alpha \to \alpha s_{0,2} \mid \alpha \in \{A, A', X, Y, Z, \dagger\}.
\end{aligned}$$

The reader can check that $L(\Pi) = L$, that is, $L \in PTL_2$.

It remains to prove that $L \notin gsm(pRE)$. Assume the contrary, and take a language $L_0 \in RE, L_0 \subseteq V^*$, and a gsm $g = (K, V, \{a,b\}, s_0, F, P)$ such that $L = g(p(L_0))$. For each string $w = aba^2b \ldots ba^{2^m}$ in L there is a string $w_0 \in p(L_0)$ such that $s_0w_0 \Longrightarrow^* ws_f$ with respect to the transitions in P and $s_f \in F$.

Assume that K contains r states and denote

$$k = \max\{|z| \mid s\alpha \to zs' \in P\}.$$

For $2^n > k \cdot (r+2)$, when translating w_0 into w, for each block a^{2^p} with $p \geq n$ there is a state s which is used at least twice. At least for such a state, the corresponding cycle introduces at least one symbol a. Consequently, for each such a block a^{2^p} there is $u \in Sub(w_0)$ and $s \in K$ such that

$$su \Longrightarrow^* a^ts, \quad \text{for some } t \geq 1.$$

Now, if m is large enough, then the same state s as above is used for two different blocks $a^{2^p}, a^{2^q}, p \neq q$. Assume that $p < q$; the case when $p > q$ is similar. That is, we can write $w_0 = w_0'uw_0''vw_0'''$ and

$$\begin{aligned}
s_0w_0'uw_0''vw_0''' &\Longrightarrow^* x'suw_0''vw_0''' \Longrightarrow^* x'a^tsw_0''vw_0'''\\
\Longrightarrow^* x'a^tx''svw_0''' &\Longrightarrow^* x'a^tx''a^{t'}sw_0''' \Longrightarrow^* x'a^tx''a^{t'}x'''s_f,
\end{aligned}$$

for some $t, t' \geq 1$, such that the string x'' contains at least one occurrence of b, and $x'a^tx''a^{t'}x''' = w$.

Clearly, also the string $z_0 = w_0'uvw_0''w_0'''$ is in $p(L_0)$, because of the permutation closure, and also the translation

$$s_0w_0'uvw_0''w_0''' \Longrightarrow^* x'a^ta^{t'}x''x'''s_f$$

is possible in g. The obtained string is of the form

$$z = aba^2b \ldots ba^{2^p+t'}b \ldots ba^{2^q-t'}b \ldots ba^{2^m}.$$

Such a string is not in L, a contradiction. Consequently, we do not have $L \in gsm(pRE)$, and this concludes the proof.

The previous theorem has a series of interesting consequences.

On the one-letter alphabet the permutation of a language is equal to the language, so $RE^{one} \subseteq PTL_2^{one}$. The inclusion $PTL \subseteq RE$ follows from Church-Turing thesis (or can be proved in a direct, constructive, manner). Consequently, we get:

Corollary 3. $RE^{one} = PTL_2^{one}$.

If we start the construction in the proof of Theorem 2 from a matrix grammar without appearance checking, then component γ_2 is useless, which implies the next result:

Corollary 4. $gsm(pMAT) \subseteq PTL_1$.

Consider the language $D_4 = \{w \in \{a_1, a_2, b_1, b_2\}^* \mid |w|_{a_i} = |w|_{b_i}, i = 1, 2\}$. It is a generator of the family of context-free languages, CF, hence each context-free language L can be written in the form $L = \gamma(D_4)$, for a gsm γ. The language D_4 is context-free and permutation closed, therefore it belongs to $pMAT$. Because $gsm(pMAT) \subseteq PTL_1$, we obtain

Corollary 5. $CF \subset PTL_1$.

We do not know whether or not the inclusion $RE \subseteq PTL_2$ (or $RE \subseteq PTL$) holds. For strictly bounded languages, such a relation is true.

Corollary 6. $RE^{bound} = PTL_2^{bound}$.

Proof. Consider a language $L \in RE, L \subseteq a_1^* \dots a_n^*$, for some $a_1, \dots, a_n \in V$, mutually different. From Theorem 2 it follows that $p(L) \in PTL_2$. We can write $L = p(L) \cap a_1^* \dots a_n^*$. An intersection with a regular language can be realized by a gsm; again from Theorem 2, we get $L \in PTL_2$. Therefore, $RE^{bound} \subseteq PTL_2^{bound}$. The converse inclusion follows from Church-Turing thesis (or can be directly proved).

6 On the State Complexity of PT Systems

The component γ_1 of the PT system in the proof of Theorem 2 has a number of states which depends on the gsm γ and the starting matrix grammar G. We do not see a way to avoid the dependence on γ. However, if we do not look for gsm images of permutation closures of languages in RE, then we can avoid the dependence on the starting grammar G: the hierarchy on the maximal number of states in the components of PT systems which generate languages in pRE collapses at the second level:

Theorem 7. *For each language $L \in pRE$ there is a PT system Π such that $L = L(\Pi)$ and each component of Π has at most two states.*

Proof. Starting from a matrix grammar $G = (N, T, S, M, C)$ in the binary normal form, with k matrices of the form $m_i : (X_i \to \alpha_i, A_i \to x_i), 1 \le i \le k$, and n matrices of the form $m_j : (X_j \to Y_j, A_j \to \dagger), k + 1 \le j \le k + n$, we construct a PT system

$$\begin{aligned}
\Pi &= (V, T, w_0, \gamma_1, \dots, \gamma_{k+n}, \gamma_{k+n+1}, \gamma_{k+n+2}),\\
V &= N_1 \cup N_2 \cup T \cup \{c, d, e, \dagger\} \cup \{\bar{a} \mid a \in T\},\\
w_0 &= XAcc, \text{ for } (S \to XA) \in M, X \in N_1, A \in N_2,
\end{aligned}$$

with the following components:

$$\begin{aligned}
\gamma_i &= (\{s_{0,i}, s_{1,i}\}, V, s_{0,i}, P_i), \text{ for } 1 \leq i \leq k, \text{ with}\\
P_i &= \{s_{0,i}X_i \to s_{1,i},\ s_{1,i}A \to \alpha_i\bar{x}_i s_{0,i},\ s_{1,i}c \to \dagger s_{1,i}\},\\
\gamma_j &= (\{s_{0,j}, s_{1,j}\}, V, s_{0,j}, P_j), \text{ for } k+1 \leq j \leq k+n, \text{ with}\\
P_j &= \{s_{0,j}X_j \to e s_{1,j},\ s_{1,j}d \to Y_j s_{0,j},\ s_{1,j}A_j \to \dagger s_{1,j}\},\\
\gamma_{k+n+1} &= (\{s_{0,k+n+1}\}, V, s_{0,k+n+1}, P_{k+n+1}),\\
P_{k+n+1} &= \{s_{0,k+n+1}e \to d s_{0,k+n+1}\}\\
&\quad \cup \{s_{0,k+n+1}\alpha \to \alpha s_{0,k+n+1} \mid \alpha \in N_1 \cup N_2 \cup \{\dagger\}\},\\
\gamma_{k+n+2} &= (\{s_{0,k+n+2}, s_{1,k+n+2}\}, V, s_{0,k+n+2}, P_{k+n+2}),\\
P_{k+n+2} &= \{s_{0,k+n+2}\bar{a} \to \bar{a}s_{0,k+n+2},\ s_{0,k+n+2}\bar{a} \to a s_{0,k+n+2} \mid a \in T\}\\
&\quad \cup \{s_{0,k+n+2}c \to s_{1,k+n+2}\}\\
&\quad \cup \{s_{1,k+n+2}\alpha \to \alpha s_{1,k+n+2} \mid \alpha \in N_1 \cup N_2 \cup \{\bar{a} \mid a \in T\} \cup \{\dagger\}\}.
\end{aligned}$$

This system works in a way similar to that in the proof of Theorem 2. The components $\gamma_i, 1 \leq i \leq k$, simulate the matrices in M which do not involve rules used in the appearance checking mode. The components $\gamma_j, k+1 \leq j \leq k+n$, simulate the matrices which contain rules used in the appearance checking mode. Note that in each moment only one of these components can work, because only one occurrence of a symbol from N_1 is present in the current multiset; moreover, the use of a rule $X_j \to Y_j$, for any i, is completed only when the simulation of the matrix in which this rule appears is completed (the symbol Y_i is introduced by a rule which returns to the initial state of the component γ_i). The component γ_{k+n+1} is used, as in the proof of Theorem 2, for ensuring the correct simulation of the matrices which contain rules used in the appearance checking manner.

The component γ_{k+n+2} is used for permuting the terminal symbols, at the end of a computation, in such a way to obtain all permutations of strings in $L(G)$ (we can wait in state $s_{0,k+n+2}$ as long as we need). Moreover, this component checks whether or not the derivation is terminal: if any nonterminal of G is present in the configuration, then we can cycle in state $s_{1,k+n+2}$.

In conclusion, $L(\Pi) = p(L(G))$.

The number of components of the system constructed in the proof of Theorem 7 depends on the starting grammar.

In all the results from this and the previous section, the length of the string x in rules $sa \to xs'$ of the components of the PT systems we have used can be bounded by two: start from a matrix grammar in the binary normal form having the string z in matrices $(X \to \alpha, A \to z)$ of length at most two (this can be arranged – see [4]). One can see from the previous constructions that the obtained PT system has the desired property.

7 Closure Properties

A way to estimate the size of a family of languages is to consider its closure properties. From this point of view, the family PTL seems to be rather large:

Theorem 8. *The family PTL is a full AFL.*

Proof. It is easy to see that the family PTL is closed under arbitrary gsm mappings; this implies the closure under arbitrary morphisms, intersection with regular languages, and inverse morphisms.

Union. Consider two PT systems $\Pi_i = (V_i, T_i, w_{0,i}, \gamma_{1,i}, \ldots, \gamma_{n_i,i}), i = 1, 2$. Without loss of generality we may assume that the states used by the components of Π_1 are different from those used by the components of Π_2 and that also $V_1 - T_1$ is disjoint of $V_2 - T_2$. We construct the system

$$\Pi = (V_1 \cup V_2 \cup \{d, d_1, d_2\}, T_1 \cup T_2, dw_{0,1}w_{0,2}, \gamma_0, \gamma'_{1,1}, \ldots, \gamma'_{n_1,1}, \gamma'_{1,2}, \ldots, \gamma'_{n_2,2}),$$

with

$$\begin{aligned}\gamma_0 &= (\{s_0\}, V_1 \cup V_2 \cup \{d, d_1, d_2\}, s_0, \{s_0 d \to d_1^{n_1} s_0,\ s_0 d \to d_2^{n_2} s_0\}),\\ \gamma'_{i,j} &= (K_{i,j} \cup \{s'_{0,i,j}\}, V_j, s'_{0,i,j}, P_{i,j} \cup \{s'_{0,i,j} d_j \to s_{0,i,j}\}),\\ &\quad \text{for all } 1 \le i \le n_j, j = 1, 2.\end{aligned}$$

One can easily see that we first work in the new component, γ_0, introducing either n_1 occurrences of d_1 or n_2 occurrences of d_2. In this way, all components of Π_1 or all components of Π_2 pass simultaneously to their initial states. From now on, these components work exactly as they are doing in the initial system. Because we have only one occurrence of d, γ_0 can work only once, hence only the components of one of Π_1, Π_2 are activated. Consequently, we get $L(\Pi) = L(\Pi_1) \cup L(\Pi_2)$.

Concatenation. Start from two systems $\Pi_i, i = 1, 2$, as above, with disjoint sets of states and sets of non-terminal symbols, and construct a new system as follows. Instead of a formal (highly cumbersome) construction, we indicate it in Figure 2 and describe it informally.

As one can see in Figure 2, we have two new components, γ_0 and γ'_0, and modified variants of all the components of Π_1 and Π_2. In particular, for each $\gamma_{i,1}, 1 \le i \le n_1$, we consider $\gamma'_{i,1}$, which "contains" $\gamma_{i,1}$ as well as a modified copy of $\gamma_{i,1}$ with all states in the form $\bar{s}$. From each state s in the copy of $\gamma_{i,1}$ to the corresponding state $\bar{s}$ in the modified copy of $\gamma_{i,1}$ we have a transition, via a rule $sd_2 \to \bar{s}$. Moreover, for each move $sa \to xs'$ from $\gamma_{i,1}$, in the modified copy we introduce the rule $\bar{s}a \to Xx\bar{s}'$.

The initial multiset is again $dw_{0,1}w_{0,2}$. At the first step, only the new component γ_0 can work; it introduces n_1 copies of d_1, which make possible the activation of all components of Π_1. These components (their copy from $\gamma'_{i,1}$) reaches their initial state and work as they work in Π_1. During this time, γ_0 can stay in state s_1, using the rule $s_1 d \to ds_1$, while all other components of the system (γ'_0 and $\gamma'_{i,2}$, for $1 \le i \le n_2$) are doing nothing, because they do not have symbols to process.

At any moment, γ_0 can introduce n_1 copies of d_2 and pass to state s_2. This is the moment when we want to finish the work of Π_1, to check whether or not this is done correctly (whether or not we have a halting configuration from the point of view of Π_1), and to pass to also simulate Π_2. This is ensured by the "controller" component γ'_0. After having introduced n_1 copies of d_2, all these copies must be used at the next step by the components $\gamma'_{i,1}, 1 \leq i \leq n_1$, otherwise the component γ'_0 can use such a symbol and enters the cycle $s'_1 d'_2 \to d'_2 s'_1$, hence the computation will never finish. A symbol d_2 can be used by a component $\gamma'_{i,1}$ by the rule $sd_2 \to \bar{s}$, where s is the current state of $\gamma_{i,1}$ reached in $\gamma'_{i,1}$ and $\bar{s}$ is a copy of it. Note that rules of the form $sd_2 \to \bar{s}$ are introduced for all states s, but only one per component can be used, because we are in a given state of each component.

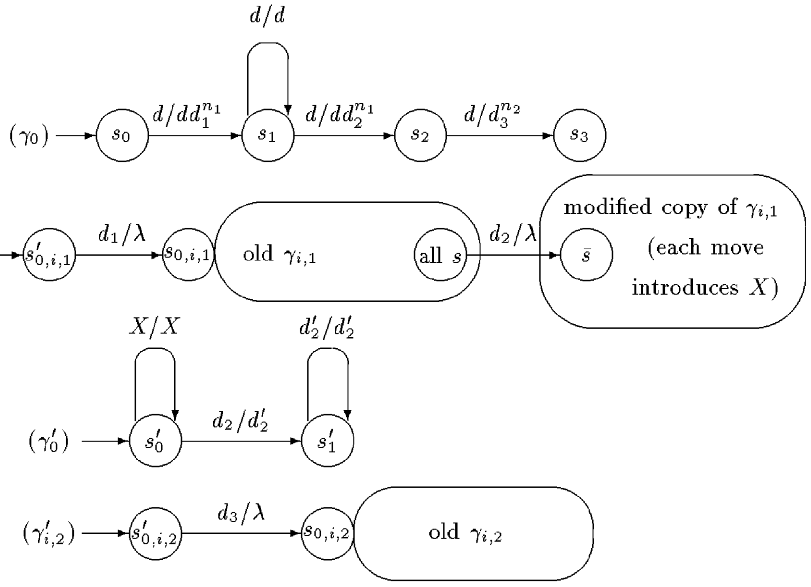


Fig. 2. The construction for concatenation.

If we are in a halting configuration of $\gamma_{i,1}$ (it is possible that such a configuration has been reached at a previous step and we have just waited for the symbol d_2 to be introduced), then we also move to a halting configuration of the copy of $\gamma_{i,1}$, that which uses barred states. If this is not the case, that is, at least one further transition can be performed in $\gamma_{i,1}$, then this is also possible in the copy of $\gamma_{i,1}$ now activated in $\gamma'_{i,1}$. Because each rule of this copy introduces an occurrence of the symbol X, the computation is again lost: the "controller" component will use this symbol X, working forever.

In conclusion, when the state s_2 is reached in γ_0, we can continue the computation without entering a cycle if and only if a halting configuration was reached in Π_1.

The component γ_0 introduces now n_2 copies of d_3, which activate the components $\gamma'_{i,2}, 1 \leq i \leq n_2$; in this way, we continue working as in Π_2, hence we obtain the concatenation of the languages $L(\Pi_1)$ and $L(\Pi_2)$.

Kleene closure. Consider a PT system $\Pi = (V, T, w_0, \gamma_1, \ldots, \gamma_n)$. We proceed as above, indicating the construction by a picture – Figure 3 – and then discussing it informally.

The new component γ_0 controls the iteration of using the system Π (in the new form, where the components γ_i were modified to $\gamma_i', 1 \le i \le n$, in a way similar to that in the proof of the closure under concatenation), and γ_0' is again the "controller" of the correct termination of a computation in Π before starting another computation.

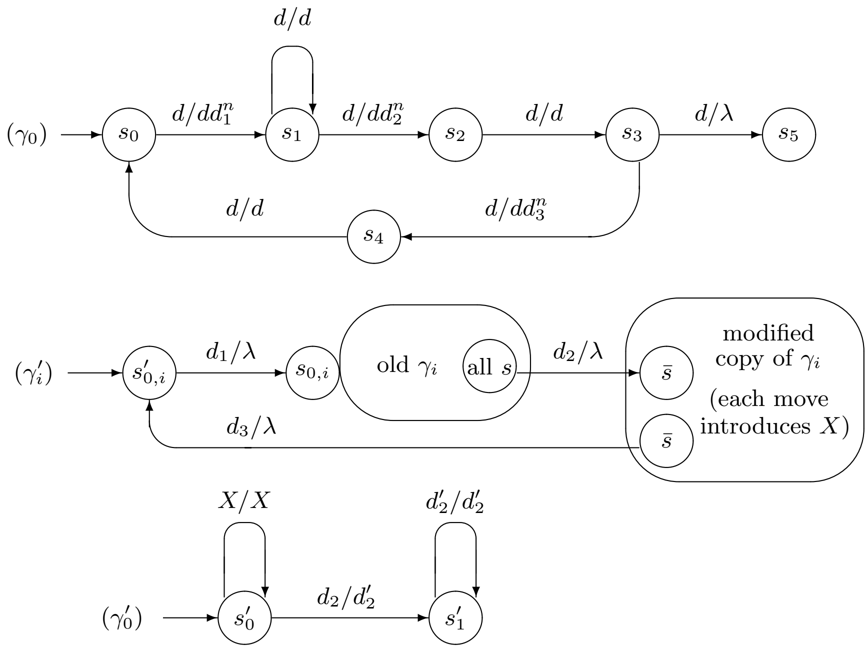


Fig. 3. The construction for Kleene +.

The initial multiset is dw_0. At the first step we can work only in γ_0, where n occurrences of d_1 are introduced. Now, each γ_i' can start working. We enter the initial states of each γ_i and then we work as in γ_i (at this time γ_0 stays in state s_1 and γ_0' stays in its initial state). At any time, γ_0 can introduce the symbol d_2 (again, n copies). The copies of d_2 should be used for passing from the current states of each γ_i to the barred version of that state in the modified copy of γ_i included in γ_i' (otherwise the computation will never stop, because γ_0' cycles in s_1'). As in the case of concatenation, if the computation in Π is not completed, then the symbol X is introduced and the computation will continue forever in γ_0'. At this time, γ_0 passes to state s_3. At the next step, each component of the system returns to its initial state, by rules of the form $\bar{s}d_3 \to s_{0,i}'$, made active by the introduction of n copies of d_3 by γ_0. At the same step, also γ_0 returns to its initial state.

In this way, we can get the concatenation of any number of strings in $L(\Pi)$. After producing any number of strings in $L(\Pi)$ (maybe only one), we can pass to state s_5 of γ_0, which ends the computation.

In conclusion, we produce $L(\Pi)^+$, which concludes the proof of the closure under Kleene + and the proof of the theorem, too.

8 Final Remarks

We have here introduced a class of computing models – called PT systems – which belongs both to Natural Computing area (Computing with Membranes, [8], [9], etc.) and to Multiset Processing ([1], [2], [3], etc.): several finite automata with outputs (gsm's) swim in a space where a multiset of symbols is present and they process these symbols in a parallel manner. We prove that such machines are rather powerful: PT systems with two components (and no bound on the number of states of each component) can generate all (and more than all) gsm images of permutation closures of recursively enumerable languages, while PT systems with an unbounded number of components, each of them having only two states, can generate all permutation closures of recursively enumerable languages.

We do not know whether or not these two parameters, the number of states and the number of components, induce a doubly infinite hierarchy of languages.

Several other problems remain to be investigated. For instance, we have considered here non-deterministic gsm's. What about using only deterministic components in our systems? Actually, we have here two types of non-determinism, one at the level of components (using non-deterministic gsm's) and one at the level of the whole system: in a given configuration, the component which takes a copy of a symbol and processes it is non-deterministically choosen among those which can do it. For instance, if we have n copies of the symbol a and $n + 1$ components can take this symbol at that moment, only n of them will work on a; the remaining component will either wait, or will use a symbol different from a, if this is possible.

A way to diminish the non-determinism at the level of the system is to consider a priority relation among components: in each moment, the components are enabled in the decreasing order of their priority.

However, even a total ordering of components does not remove completely the non-determinism: if in a given state of a gsm we can read both a and b, and these symbols are present in the current multiset, then we may choose one of the two possible steps (note that this does not appear when gsm's translate strings, because in that case at each moment only one symbol is scanned by the read head).

On the other hand, systems which are *completely deterministic* (at each moment, only one next configuration is possible) do not seem to be of much interest: they can proceed along a unique computation, which either stops (hence the generated language is empty or finite), or continues forever (hence the language is empty).

The study of determinism in PT systems, at various levels, deserves a further investigation.

References

1. J. P. Banâtre, A. Coutant, D. Le Metayer, A parallel machine for multiset transformation and its programming style, *Future Generation Computer Systems*, **4** (1988), 133–144.
2. J. P. Banâtre, D. Le Metayer, Programming by multiset transformation, *Communications of the ACM*, **36** (1993), 98–111.
3. G. Berry, G. Boudol, The chemical abstract machine, *Theoretical Computer Sci.*, **96** (1992), 217–248.
4. J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer, Berlin, 1989.
5. J. Dassow, Gh. Păun, On the power of membrane computing, *J. Univ. Computer Sci.*, **5**, 2 (1999), 33–49.
6. A. Kelemenova, J. Kelemen, A grammar-theoretic treatment of multiagent systems, *Cybernetics and Systems*, **23** (1992), 210–218.
7. S. N. Krishna, R. Rama, A variant of P systems with active membranes: Solving NP-complete problems, *Romanian J. of Information Science and Technology*, **2**, 4 (1999).
8. Gh. Păun, Computing with membranes, submitted, 1998 (see also *TUCS Research Report* No. 208, November 1998, http://www.tucs.fi).
9. Gh. Păun, Computing with membranes. An introduction, *Bulletin of the EATCS*, **67** (1999), 139–152.
10. Gh. Păun, P systems with active membranes: Attacking NP complete problems, submitted 1999, and *Auckland University, CDMTCS Report* No 102, 1999 (www.cs.auckland.ac.nz/CDMTCS).
11. Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing. New Computing Paradigms*, Springer-Verlag, Berlin, 1998.
12. Gh. Păun, G. Rozenberg, A. Salomaa, Membrane computing with external output, submitted, 1998 (see also *TUCS Research Report* No. 218, December 1998, http://www.tucs.fi).
13. Gh. Păun, T. Yokomori, Membrane computing based on splicing, *Preliminary Proc. of Fifth Intern. Meeting on DNA Based Computers* (E. Winfree, D. Gifford, eds.), MIT, June 1999, 213–227.
14. Gh. Păun, S. Yu, On synchronization in P systems, *Fundamenta Informaticae*, **38**, 4 (1999), 397–410,
15. I. Petre, A normal form for P systems, *Bulletin of the EATCS*, **67** (1999), 165–172.
16. G. Rozenberg, A. Salomaa, Eds., *Handbook of Formal Languages*, 3 volumes, Springer-Verlag, Berlin, 1997.
17. Y. Suzuki, H. Tanaka, On a LISP implementation of a class of P systems, submitted, 1999.

A Structural Method for Output Compaction of Sequential Automata Implemented as Circuits

M. Seuring and M. Gössel

University of Potsdam, Institute of Computer Science
Fault Tolerant Computing Group
14415 Potsdam, Germany
markus@mpag-inf.uni-potsdam.de, mgoessel@mpag-inf.uni-potsdam.de

Abstract. In this paper output space compaction for sequential circuits is considered for the first time. Based on simple estimates for the probabilities of the existence of sensitized paths from the signal lines to the circuit outputs, optimal output partitions can be determined without fault simulation. The outputs are partitioned in such a way that internal stuck-at faults influence at most one of the outputs of a group with high probability. The proposed method is primarily developed for concurrent checking. On average with less than 4 compacted groups of outputs an error detection probability of 98% can be achieved. As the experimental results show, the method is also effectively applicable in pseudo-random test mode. On average for three groups of compacted outputs there is no reduction of the fault coverage for a pseudo-random off-line test. Since the proposed algorithm is of linear complexity with respect to the number of circuit lines and of quadratic complexity with respect to the number of primary circuit outputs large automata can be efficiently processed.

1 Introduction

As the complexity of VLSI continues to increase, the number of inputs/outputs of automata which are implemented on the IC as sequential circuits are increasing accordingly. For circuits with a large number of outputs methods of output space compaction are of growing interest. These methods allow to decrease the number of observed outputs and therefore to reduce the necessary hardware overhead for concurrent checking and testing.

Until now different methods for output space compaction were considered for combinational circuits only. The results obtained in [2], [3], [4], [5], [7], [8] are applicable in test mode. Output space compaction for concurrent checking for combinational circuits was investigated in [6], [9], [10]. In this paper for the first time output space compaction for synchronous sequential circuits for concurrent checking is considered. The proposed method is an extension of the structural method for output space compaction for combinational circuits described in [9].

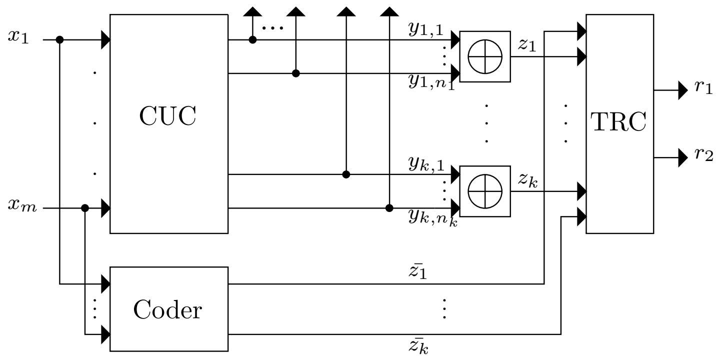


Fig. 1. Linear output compaction for concurrent checking

2 Linear Output Compaction for Concurrent Checking

Linear output space compaction for concurrent checking is illustrated in Fig. 1. The circuitry of Fig. 1 consists of the monitored circuit or the circuit under check (CUC) with n primary outputs, a coder and a two-rail checker (TRC). The primary outputs are divided into k disjoint groups. The outputs of every group are compacted by an XOR-tree into the k compacted outputs. Each compacted output represents the parity of the corresponding group. The coder generates the k inverted compacted outputs. The outputs of the coder are compared with the compacted outputs of the CUC by the (self checking) two-rail checker TRC. An error signal of the two-rail checker indicates a fault of the CUC, the coder, an XOR-tree, or the (self-checking) two-rail checker TRC.

For systems with a very large number of outputs, linear output space compaction, which is widely used in fault tolerant system design, can be used to simplify the method of duplication and comparison. Instead of comparing all the outputs of the duplicated circuits using a huge comparator, only a small number of properly compacted outputs of the duplicated systems need to be compared. Nearly the same fault coverage can be achieved.

3 Error Propagation in Combinational Circuits

In this section we briefly describe how the probabilities that errors of the circuit lines of a combinational circuit are propagated to each of the different circuit outputs can be simply approximated. The described algorithm is of linear complexity with respect to the number of circuit lines. The linear complexity is of great importance for an efficient processing of large circuits.

We consider in this section a combinational circuit C with m inputs $x_1, \ldots, x_m$ and n outputs $y_1, \ldots, y_n$ which is given as a netlist of AND-, OR-, NAND-, NOR-, XOR-, XNOR-gates and INVERTERS. We denote an error at a signal line s of C by $e(s)$ and the set of all signal lines of C by L_C.

As a simplification we assume that the values of all signal lines $s \in L_C$ (including the input lines of C) are randomly chosen and equally distributed with the probabilities $p(s{=}1) = p(s{=}0) = \frac{1}{2}$. Under this assumption we determine for every circuit line s and for every circuit output y a simple approximation $p(s \rightsquigarrow y)$ of the probability that there is a sensitized path from s to y. This probability is an estimate that an error $e(s)$ at a circuit line s results in an erroneous circuit output y.

Now we explain how the corresponding probabilities for the existence of sensitized paths can be approximated. At first we consider a single gate G with an erroneous input value. This erroneous value is propagated to the output of the gate if the values of all the other input lines of G are non-controlling values[1]. If we assume that the value of each signal line s is randomly chosen with the probabilities $p(s{=}1) = p(s{=}0) = \frac{1}{2}$ then the probability that the error is propagated to the output of an AND, OR, NAND, or NOR-gate with k input lines is $p(G) = 1/(2^{k-1})$. Since both the values 0 and 1 are non-controlling values of XOR- and XNOR-gates we have $p(G) = 1$ for these gates. For an INVERTER we also have $p(G) = 1$.

For a combinational circuit C with n outputs to every circuit line s, an n-dimensional vector $p_s = (p(s \rightsquigarrow y_1), \ldots, p(s \rightsquigarrow y_n))$ is assigned. The components $p(s \rightsquigarrow y_1), \ldots, p(s \rightsquigarrow y_n)$ are the approximated probabilities that paths from the line s to the circuit outputs $y_1, \ldots, y_n$ are sensitized.

These vectors are computed by passing through the circuit from the outputs to the inputs in reverse topological order and under the assumption that the probability of an error to be propagated by a gate G is $p(G)$. For more details see [9].

4 Error Propagation in Sequential Circuits

We consider now a sequential circuit S consisting of a combinational part C_S, p flip-flops $R_1, \ldots, R_p$, m primary inputs $x_1, \ldots, x_m$, and n primary outputs $y_1, \ldots, y_n$. S is supposed to be given as a netlist of gates and the set of all signal lines is denoted by L_S.

An error $e(s)$ of a signal line can be propagated to the primary outputs either directly during the same clock cycle or with a delay of N, $N \geq 1$, clock cycles passing the flip flop elements of the sequential circuit N-times. As the delay N increases, the lengths of the corresponding paths increase and the probability that one of these paths is sensitized decreases. Thereby the length of a path is determined by the number of AND-, NAND-, OR- or NOR-gates on this path. The approximated probability that a path is sensitized exponentially decreases with its length and therefore also with the corresponding delay N. In accordance with these considerations the proposed algorithm for the determination of the approximated probabilities restricts the considered paths to paths with a maximal delay N_{max}. As the experimental results show the value $N_{max} = 8$ guarantees the necessary accuracy.

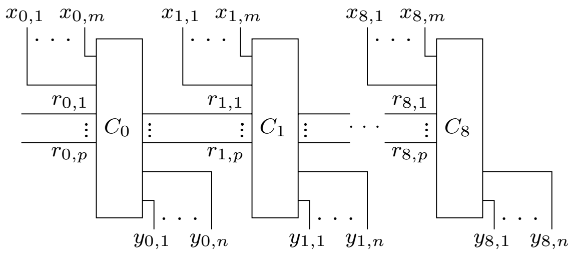


Fig. 2. Iterative array of time frames

The sequential circuit S is modeled as an iterative array A_S of $N_{max}+1 = 8+1 = 9$ time frames $C_0, C_1, \ldots, C_8$ as described, for instance, in [1]. The corresponding iterative array of 9 time frames is shown in Fig. 2. The combinational parts C_i, $0 \leq i \leq 8$ are identical to the combinational part C of the sequential circuit. The inputs and the outputs of the time frame C_i are denoted by $x_{i,1}, \ldots, x_{i,m}$ and $y_{i,1}, \ldots, y_{i,n}$, respectively. A signal line s of the original sequential circuit S corresponds to 9 signal lines $s_0, s_1, \ldots, s_8$ within the time frames $C_0, C_1, \ldots, C_9$ respectively.

The approximated probability $p(s \rightsquigarrow y_{i,j})$ that a sensitized path exists from line s_0 of C_0 to the j-th output of the i-th time frame C_i is computed by applying an extension of the algorithm described in Section 3.

5 Output Space-Compaction

In this section we describe how the groups of outputs which are compacted by XOR-trees can be efficiently determined. If exactly two of the circuit outputs are simultaneously erroneous this error can only be detected if the erroneous outputs are in different groups. Therefore, if the probability that two outputs y and y' are simultaneously erroneous for a given fault is high then these outputs should be in different groups of compacted outputs.

For the sequential circuit S the approximate probability $p_S(s \rightsquigarrow y_j)$ that an error $e(s)$ is propagated from the signal line s to the primary output y_j is computed by

$$p_S(s \rightsquigarrow y_j) = \sum_{i=0}^{N_{max}} p(s_0 \rightsquigarrow y_{i,j}).$$

The signal line s_0 in the time frame C_0 of A_S corresponds to the line s of the sequential circuit S, and $p(s \rightsquigarrow y_{i,j})$ is the probabilities that an error $e(s_0)$ in C_0 is propagated to the j-th output of the i-th time frame C_i, $1 \leq i \leq N_{max}$. As we have already pointed out we chose $N_{max} = 8$.

Now we compute the approximation $P(y, y')$, $y \neq y'$, of the probability that a randomly chosen signal line of S is erroneous and this error is detectable at both primary outputs y and y' of S. As a simplification we assume that for all

$s \in S$ the probabilities $p_S(s \rightsquigarrow y)$ and $p_S(s \rightsquigarrow y')$ for $y \neq y'$ are independent. Then we obtain

$$P(y, y') = \frac{1}{|L_S|} \sum_{s \in L_S} (p_S(s \rightsquigarrow y) \cdot p_S(s \rightsquigarrow y')).$$

Using these probabilities $P(y, y')$ we now describe a heuristic for the determination of a partition $\mathcal{G}_k$ of the set Y into k disjoint groups $G_1, \ldots, G_k$ with $G_1 \cup \ldots \cup G_k = Y$, $G_i \cap G_j = \emptyset$ for $i \neq j$.

The outputs of every group G_i, $1 \le i \le k$ are compacted by an XOR-tree into the compacted group output z_i. For a given k the partition $\mathcal{G}_k$ is computed in such a way that the sum

$$P(\mathcal{G}_k) = \sum_{G_i \in \mathcal{G}_k} \left(\sum_{\substack{y_s, y_t \in G_i \\ s<t}} P(y_s, y_t) \right)$$

is minimized.

If the probability $P(y_s, y_t)$ is small for all pairs y_s, y_t of outputs which are both elements of the same group G_i then the value $P(\mathcal{G}_k)$ of the partition $\mathcal{G}_k$ is also small.

For grouping the outputs first we assign to every output $y_j \in Y$ a connection weight $W(y_j)$,

$$W(y_j) = \sum_{y \in Y \setminus \{y_j\}} P(y_j, y).$$

Then we sort the circuit outputs in descending order of their connection weights. We assign the first k outputs to the k different groups $G_1, \ldots, G_k$. The remaining outputs are assigned according to the following rule: If l, $k \le l < n$ outputs are already assigned to the groups $G_1, \ldots, G_k$ then we assign the next output y to that group $G_j \in \{G_1, \ldots, G_k\}$ for which the sum

$$w_j = \sum_{y' \in G_j} P(y, y').$$

is minimal.

The proposed algorithm for output space-compaction is linear with respect to the number of signal lines or the number of gates and quadratic with respect to the number of circuit outputs. Thus, the proposed algorithm can be applied to large sequential circuits. For a given number k of groups of compacted outputs optimal partitions can be determined without fault simulation.

6 Experimental Results

Experimental results are derived for 20 benchmark circuits of the 1989 International Symposium on Circuits and Systems. For each of the benchmark circuits and for $k = 2, \ldots, 8$ the partition $\mathcal{G}_k$ of their outputs are determined as described in the previous section. The necessary CPU-time of a SUN SPARC 5 is less than one second if the number of gates is limited to 1000. The largest investigated circuit is s15850.1 with 9785 gates and 534 flipflops and the corresponding run time is approximately 8 seconds. These times include the computation of the probabilities $p(s \rightsquigarrow y)$ for $N = 8$ for all internal lines s, and for all circuit outputs y. The trivial partition $\mathcal{G}_1$, i.e. the compaction of all circuit outputs by a single XOR-tree into the parity of the circuit outputs is also considered.

Experimental results are obtained with respect to single stuck-at faults, transient faults, and intermittent faults for concurrent checking in on-line mode. For on average 3.7 groups at least 98% of the errors which are detected at the outputs of S are detected in the same clock cycle at the compacted outputs of S_k.

Although the method was developed for concurrent checking it can also successfully be applied for pseudo-random off-line testing. On average less than 3 groups of compacted outputs guarantee that no reduction of the fault coverage with respect to the original circuit in a pseudo-random off-line test is obtained.

Since the proposed algorithm is of linear complexity with respect to the number of circuit lines and of quadratic complexity with respect to the number of primary circuit outputs large circuits can be efficiently processed.

References

1. M. Abramovici, M. A. Breuer, A. D. Friedman, “Digital Systems Testing and Testable Design,” IEEE Press, 1990.
2. P. Böhlau, “Zero Aliasing Compression Based on Groups of Weakly Independent Outputs in Circuits with High Complexity for Two Fault Models,” in *Lect. Notes in Comp. Science,* vol. 852, pp. 289-306, 1994, Springer-Verlag.
3. K. Chakrabarty and J. P. Hayes, “Test Response Compaction Using Multiplexed Parity Trees,” in *IEEE Trans. Computer-Aided Design,* vol. 15, pp. 1399-1408, Nov. 1996.
4. K. Chakrabarty and J. P. Hayes, “Zero-Aliasing Space Compaction of test Responses Using Multiple Parity Signatures,” in *IEEE Trans. VLSI Systems,* vol. 6, no. 2, pp. 309-313, Nov. 1996.
5. K. Chakrabarty, B. T. Murray, J. P. Hayes, “Optimal Space Compaction of Test Responses,” in *Proc. Int. Test Conf.,* pp. 834-843, 1995.
6. M. Gössel and E. S. Sogomonyan, “Design of Self-Testing and On-Line Fault Detection Combinational Circuits with Weakly Independent Outputs,” in *J. Electronic testing: Theory and Applications,* vol. 4, pp. 267-281, 1993.
7. A. Ivanov, B. K. Tsuji, Y. Zorian, “Programmable BIST Space Compactors,” in *IEEE Trans. Computers,* vol. 45, pp. 1393-1404, Dec. 1996.
8. J. Savir, “Shrinking Wide Compressors,” in *IEEE Trans. Computer-Aided Design,* vol. 14, pp. 1379-1387, Nov. 1995.
9. M. Seuring, M. Gössel, E. Sogomonyan, “A Structural Approach for Space Compaction for Concurrent Checking and BIST,” in *IEEE VLSI Test Symp.,* pp. 354-361, April 1998.
10. E. S. Sogomonyan, “Design of Built-in Self-Checking Monitoring Circuits for Combinational Devices,” in *Automation and Remote Control,* vol. 35(2), pp. 280-289, 1974.

An Algorithm to Verify Local Threshold Testability of Deterministic Finite Automata

A.N. Trahtman

Bar-Ilan University, Dep. of Math. and CS, 59200,Ramat Gan, Israel
trakht@macs.biu.ac.il

Abstract. A locally threshold testable language L is a language with the property that for some nonnegative integers k and l, whether or not a word u is in the language L depends on (1) the prefix and suffix of the word u of length $k-1$ and (2) the set of intermediate substrings of length k of the word u where the sets of substrings occurring *at least* j times are the same, for $j \leq l$. For given k and l the language is called l-threshold k-testable. A finite deterministic automaton is called l-threshold k-testable if the automaton accepts a l-threshold k-testable language.
In this paper, the necessary and sufficient conditions for an automaton to be locally threshold testable are found. We introduce the first polynomial time algorithm to verify local threshold testability of the automaton based on this characterization.
New version of polynomial time algorithm to verify the local testability will be presented too.

Keywords: *deterministic finite automaton, locally threshold testable, algorithm, semigroup*

AMS subject classification 68Q25, 68Q45, 68Q68, 20M07

Introduction

The concept of local testability was introduced by McNaughton and Papert [18] and by Brzozowski and Simon [7]. Local testability can be considered as a special case of local l-threshold testability for $l = 1$. Locally testable languages, automata and semigroups have been investigated from different points of view (see [6] - [15], [17], [20], [25] - [32]). In [19], local testability was discussed in terms of "diameter-limited perceptrons". Locally testable languages are a generalization of the definite and reverse-definite languages, which can be found, for example, in [10] and [21]. Some variations of the concept of local testability (strictly, strongly) obtained by changing or omitting prefixes and suffixes in the definition of the concept were studied in [6], [8], [18], [20], [25].

Locally testable automata have a wide spectrum of applications. Regular languages and picture languages can be described by a strictly locally testable languages [6], [12]. Local automata (a kind of locally testable automata) are heavily used to construct transducers and coding schemes adapted to constrained channels [3]. Literal morphisms may be modelled by help of 2-testable languages [9].

In [11], locally testable languages are used in the study of DNA and informational macromolecules in biology.

Kim, McNaughton and McCloskey ([13], [14]) have found necessary and sufficient conditions of local testability and a polynomial time algorithm for local testability problem based on these conditions. The realization of the algorithm is described by Caron in [8]. A polynomial time algorithm for local testability problem for semigroups was presented in [26].

The locally threshold testable languages were introduced by Beauquier and Pin [4]. These languages generalize the concept of locally testable language and have been studied extensively in recent years (see [5], [20], [24], [29], [30]). The syntactic characterization of locally threshold testable languages one can find in [4]:

Given the syntactic semigroup S of the language L, we form a graph G(S) as follows. The vertices of G(S) are the idempotents of S, and the edges from e to f are the elements of the form esf. A language L is locally threshold testable if and only if S is aperiodic and for any two nodes e, f and three edges p, q, r such that p and q are edges from e to f and r is an edge from f to e we have

$$\mathrm{prq}=\mathrm{qrp}$$

Since only five elements of the semigroup S are considered, there exists a polynomial time algorithm of order $O(|S|^5)$ for local threshold testability problem in the case of semigroups. But the cardinality of the syntactic semigroup of a locally threshold testable automaton is not polynomial in the number of its nodes [14]. This is why the study of the automaton and the state transition graph of the automaton is important from the practical point of view (see [14], [15]) and we use here this approach.

For the state transition graph Γ of an automaton, we consider some subgraphs of the cartesian product $\Gamma \times \Gamma$ and $\Gamma \times \Gamma \times \Gamma$. In this way, necessary and sufficient conditions for a deterministic finite automaton to be locally threshold testable are found. We present here $O(n^5)$ time algorithm to verify local threshold testability of the automaton based on this characterization.

Necessary and sufficient conditions of local testability from [14] are considered in this paper in terms of reachability in the graph $\Gamma \times \Gamma$. New version of $O(n^2)$ time algorithm to verify local testability based on this approach will be presented too.

Notation and Definitions

Let Σ be an alphabet and let Σ^+ denote the free semigroup on Σ. If $w \in \Sigma^+$, let $|w|$ denote the length of w. Let k be a positive integer. Let $i_k(w)$ $[t_k(w)]$ denote the prefix [suffix] of w of length k or w if $|w| < k$. Let $F_{k,j}(w)$ denote the set of factors of w of length k with at least j occurrences. A language L [a semigroup S] is called **l-threshold k-testable** if there is an alphabet Σ [and a surjective morphism $\phi : \Sigma^+ \to S$] such that for all u, $v \in \Sigma^+$, if $i_{k-1}(u) = i_{k-1}(v)$, $t_{k-1}(u) = t_{k-1}(v)$ and $F_{k,j}(u) = F_{k,j}(v)$ for all $j \leq l$, then either both u and v are in L or neither is in L $[u\phi = v\phi]$.

An automaton is **l-threshold k-testable** if the automaton accepts a l-threshold k-testable language [the syntactic semigroup of the automaton is **l-threshold k-testable**].

A language L [a semigroup S, an automaton $\mathbf{A}$] is **locally threshold testable** if it is l-threshold k-testable for some k and l.

A semigroup without non-trivial subgroups is called **aperiodic** [4]

$|\Gamma|$ denotes the number of nodes of the graph Γ.

Γ^i denotes the direct product of i copies of the graph Γ.

A maximal strongly connected component of the graph will be denoted for brevity as **SCC** [13], a finite deterministic automaton will be denoted as **DFA** [14]. A node from an SCC will be called for brevity as an **SCC − node**.

If an edge $\mathbf{p} \to \mathbf{q}$ is labeled by σ then let us denote the node $\mathbf{q}$ as $\mathbf{p}\sigma$.

We shall write $\mathbf{p} \succeq \mathbf{q}$ if the node $\mathbf{q}$ is reachable from the node $\mathbf{p}$ or $\mathbf{p} = \mathbf{q}$.

In the case $\mathbf{p} \succeq \mathbf{q}$ and $\mathbf{q} \succeq \mathbf{p}$ we write $\mathbf{p} \sim \mathbf{q}$ (that is $\mathbf{p}$ and $\mathbf{q}$ belong to one SCC or $\mathbf{p} = \mathbf{q}$).

1 The Necessary and Sufficient Conditions

Let us formulate the result of Beauquier and Pin [4] in the following form:

Theorem 11 *[4] A language L is locally threshold testable if and only if the syntactic semigroup S of L is aperiodic and for any two idempotents e, f and elements a, u, b of S we have*

$$eafuebf = ebfueaf \tag{1}$$

Let us recall the concept of implicit operation [22], [2]: The unary operation x^ω assigns to every element x of a finite semigroup the unique idempotent in the subsemigroup generated by x.

The set of locally threshold testable semigroups forms a pseudovariety of semigroups ([29], [4]). So the theorem 11 implies

Corollary 12 *The pseudovariety of locally threshold testable semigroups consists of aperiodic semigroups and satisfies the pseudoidentity*

$$x^\omega yz^\omega ux^\omega tz^\omega = x^\omega tz^\omega ux^\omega yz^\omega$$

Lemma 13 *Let the node $(\mathbf{p}, \mathbf{q})$ be an SCC-node of Γ^2 of a locally threshold testable DFA with state transition graph Γ and suppose that $\mathbf{p} \sim \mathbf{q}$.*

Then $\mathbf{p} = \mathbf{q}$.

Proof. The transition semigroup S of the automaton is finite and aperiodic [16]. Suppose that for some element $e \in S$ and for some states $\mathbf{q}$ and $\mathbf{p}$ from SCC X we have $\mathbf{q}e = \mathbf{q}$ and $\mathbf{p}e = \mathbf{p}$. In view of $\mathbf{q}e^i = \mathbf{q}$, $\mathbf{p}e^i = \mathbf{p}$ and finiteness of S we can assume e is an idempotent. In the SCC X for some a, b from S we have

$\mathbf{p}a = \mathbf{q}$ and $\mathbf{q}b = \mathbf{p}$. Hence, $\mathbf{p}eae = \mathbf{q}$, $\mathbf{q}ebe = \mathbf{p}$. So $\mathbf{p}eaebe = \mathbf{p} = \mathbf{p}(eaebe)^i$ for any integer i. There exists a natural number n such that in the aperiodic semigroup S we have $(eae)^n = (eae)^{n+1}$. From theorem 11 it follows that for the idempotent e, $eaeebe = ebeeae$. We have $\mathbf{p} = \mathbf{p}eaebe = \mathbf{p}(eaeebe)^n = \mathbf{p}(eae)^n(ebe)^n = \mathbf{p}(eae)^{n+1}(ebe)^n = \mathbf{p}(eae)^n(ebe)^n eae = \mathbf{p}eae = \mathbf{q}$. So $\mathbf{p} = \mathbf{q}$. □

Theorem 14 *For DFA* $\mathbf{A}$ *with state transition graph* Γ *the following three conditions are equivalent:*

1)$\mathbf{A}$ *is locally threshold testable.*

2)If the nodes $(\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)$ *and* $(\mathbf{q}, \mathbf{r}, \mathbf{t}_1, \mathbf{t})$ *are SCC-nodes of* Γ^3 *and* Γ^4*, correspondingly, and*

$(\mathbf{q}, \mathbf{r}) \succeq (\mathbf{q}_1, \mathbf{r}_1)$*,* $(\mathbf{p}, \mathbf{q}_1) \succeq (\mathbf{r}, \mathbf{t})$*,* $(\mathbf{p}, \mathbf{r}_1) \succeq (\mathbf{q}, \mathbf{t}_1)$ *holds in* Γ^2

then $\mathbf{t} = \mathbf{t}_1$*.*

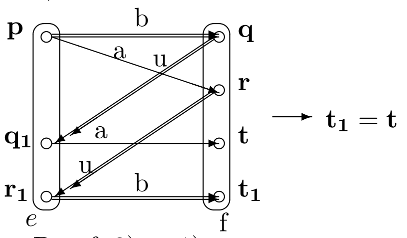
3)If the node $(\mathbf{u}, \mathbf{v})$ *is an SCC-node of the graph* Γ^2 *and* $\mathbf{u} \sim \mathbf{v}$ *then* $\mathbf{u} = \mathbf{v}$*.*

If the nodes $(\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)$*,* $(\mathbf{q}, \mathbf{r}, \mathbf{t})$*,* $(\mathbf{q}, \mathbf{r}, \mathbf{t}_1)$ *are SCC-nodes of the graph* Γ^3 *and*

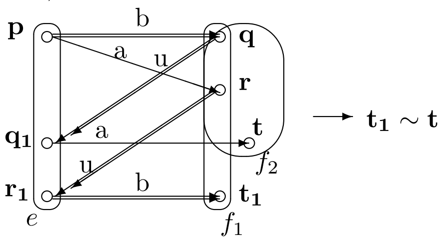
$(\mathbf{q}, \mathbf{r}) \succeq (\mathbf{q}_1, \mathbf{r}_1)$*,* $(\mathbf{p}, \mathbf{q}_1) \succeq (\mathbf{r}, \mathbf{t})$*,* $(\mathbf{p}, \mathbf{r}_1) \succeq (\mathbf{q}, \mathbf{t}_1)$ *hold in* Γ^2*,*

then $\mathbf{t} \sim \mathbf{t}_1$*.*

2)



3)



Proof. 2) $\to$ 1):

Let us consider the nodes $\mathbf{z}ebfueaf$ and $\mathbf{z}eafuebf$ where $\mathbf{z}$ is an arbitrary node of Γ, a, u, b are arbitrary elements from transition semigroup S of the automaton and e, f are arbitrary idempotents from S. Let us denote

$\mathbf{z}e = \mathbf{p}$, $\mathbf{z}ebf = \mathbf{q}$, $\mathbf{z}eaf = \mathbf{r}$, $\mathbf{z}eafue = \mathbf{r}_1$, $\mathbf{z}ebfue = \mathbf{q}_1$, $\mathbf{z}ebfueaf = \mathbf{t}$, $\mathbf{z}eafuebf = \mathbf{t}_1$.

By condition 2), we have $\mathbf{t} = \mathbf{t}_1$, whence $\mathbf{z}ebfueaf = \mathbf{z}eafuebf$. Thus, the condition $eafuebf = ebfueaf$ (1) holds for the transition semigroup S. By theorem 11, the automaton is locally threshold testable.

1) $\to$ 3):

If the node $(\mathbf{u}, \mathbf{v})$ belongs to some SCC of the graph Γ^2 and $\mathbf{u} \sim \mathbf{v}$ then by lemma 13 local threshold testability implies $\mathbf{u} = \mathbf{v}$.

The condition $eafuebf = ebfueaf$ ((1), theorem 11) holds for the transition semigroup S of the automaton. Let us consider nodes $\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{t}, \mathbf{q}_1, \mathbf{r}_1, \mathbf{t}_1$ satisfying the condition 3). Suppose

$$(\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)e = (\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1),\ (\mathbf{q}, \mathbf{r}, \mathbf{t})f_2 = (\mathbf{q}, \mathbf{r}, \mathbf{t}),\ (\mathbf{q}, \mathbf{r}, \mathbf{t}_1)f_1 = (\mathbf{q}, \mathbf{r}, \mathbf{t}_1)$$

for some idempotents $e, f_1, f_2 \in S$, and

$$(\mathbf{p}, \mathbf{q}_1)a = (\mathbf{r}, \mathbf{t}),\ (\mathbf{p}, \mathbf{r}_1)b = (\mathbf{q}, \mathbf{t}_1)\ (\mathbf{q}, \mathbf{r})u = (\mathbf{q}_1, \mathbf{r}_1)$$

for some elements $a, b, u \in S$. Then $\mathbf{p}eaf_2 = \mathbf{p}eaf_1$ and $\mathbf{p}ebf_2 = \mathbf{p}ebf_1$.

We have $\mathbf{t}_1 f_2 = \mathbf{p}eaf_1uebf_1f_2$. By theorem 11, $\mathbf{p}ebf_jueaf_j = \mathbf{p}eaf_juebf_j$ for $j = 1, 2$. So we have $\mathbf{t}_1 f_2 = \mathbf{p}eaf_1uebf_1f_2 = \mathbf{p}ebf_1ueaf_1f_2$. In view of $\mathbf{p}ebf_2 = \mathbf{p}ebf_1$ and $f_i = f_if_i$ we have $\mathbf{t}_1f_2 = \mathbf{p}ebf_2f_2ueaf_1f_2$. By theorem 11, $\mathbf{t}_1f_2 = \mathbf{p}e(bf_2)f_2ue(af_1)f_2 = \mathbf{p}e(af_1)f_2ue(bf_2)f_2$. Now in view of $\mathbf{p}eaf_2 = \mathbf{p}eaf_1$ let us exclude f_1 and obtain $\mathbf{t}_1f_2 = \mathbf{p}eaf_2uebf_2 = \mathbf{t}$. So $\mathbf{t}_1f_2 = \mathbf{t}$. Analogously, $\mathbf{t}f_1 = \mathbf{t}_1$.

Hence, $\mathbf{t}_1 \sim \mathbf{t}$. Thus 3) is a consequence of 1).

3) $\rightarrow$ 2):

Suppose that $(\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)e = (\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)$, $(\mathbf{q}, \mathbf{r}, \mathbf{t}, \mathbf{t}_1)f = (\mathbf{q}, \mathbf{r}, \mathbf{t}, \mathbf{t}_1)$, for some idempotents e, f from transition semigroup S of the automaton and

$$(\mathbf{p}, \mathbf{q}_1)a = (\mathbf{r}, \mathbf{t}),\ (\mathbf{p}, \mathbf{r}_1)b = (\mathbf{q}, \mathbf{t}_1),\ (\mathbf{q}, \mathbf{r})u = (\mathbf{q}_1, \mathbf{r}_1)$$

for some elements a, u, $b \in S$. Therefore

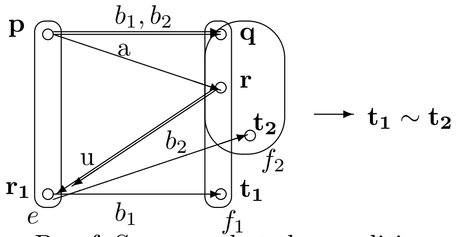
$$(\mathbf{p}, \mathbf{q}_1)eaf = (\mathbf{p}, \mathbf{q}_1)af = (\mathbf{r}, \mathbf{t})$$
$$(\mathbf{p}, \mathbf{r}_1)ebf = (\mathbf{p}, \mathbf{r}_1)bf = (\mathbf{q}, \mathbf{t}_1)$$
$$(\mathbf{q}, \mathbf{r})u = (\mathbf{q}, \mathbf{r})fue = (\mathbf{q}_1, \mathbf{r}_1)$$

for idempotents e, f and elements a, u, $b \in S$.

For $f = f_1 = f_2$ from 3) we have $\mathbf{t} \sim \mathbf{t}_1$. Notice that $(\mathbf{t}_1, \mathbf{t})f = (\mathbf{t}_1, \mathbf{t})$. The node $(\mathbf{t}_1, \mathbf{t})$ belongs to some SCC of the graph Γ^2 and $\mathbf{t} \sim \mathbf{t}_1$, whence by by lemma 13, $\mathbf{t} = \mathbf{t}_1$. $\square$

Lemma 15 *Let the nodes* $(\mathbf{q}, \mathbf{r}, \mathbf{t}_1)$ *and* $(\mathbf{q}, \mathbf{r}, \mathbf{t}_2)$ *be SCC-nodes of the graph* Γ^3 *of a locally threshold testable DFA with state transition graph* Γ. *Suppose that* $(\mathbf{p}, \mathbf{r_1}) \succeq (\mathbf{q}, \mathbf{t}_1)$, $(\mathbf{p}, \mathbf{r_1}) \succeq (\mathbf{q}, \mathbf{t}_2)$ *in the graph* Γ^2 *and* $\mathbf{p} \succeq \mathbf{r} \succeq \mathbf{r}_1$.

Then $\mathbf{t}_1 \sim \mathbf{t}_2$.



Proof. Suppose that the conditions of the lemma hold but $\mathbf{t}_1 \not\sim \mathbf{t}_2$.

We have $(\mathbf{p}, \mathbf{r}_1)e = (\mathbf{p}, \mathbf{r}_1)$, $(\mathbf{q}, \mathbf{r}, \mathbf{t}_1)f_1 = (\mathbf{q}, \mathbf{r}, \mathbf{t}_1)$, $(\mathbf{q}, \mathbf{r}, \mathbf{t}_2)f_2 = (\mathbf{q}, \mathbf{r}, \mathbf{t}_2)$, for some idempotents e, f_2, f_2 from the transition semigroup S of the automaton and

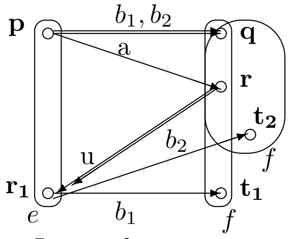
$$(\mathbf{p}, \mathbf{r}_1)b_1 = (\mathbf{q}, \mathbf{t}_1),\ (\mathbf{p}, \mathbf{r}_1)b_2 = (\mathbf{q}, \mathbf{t}_2),\ \mathbf{p}a = \mathbf{r},\ \mathbf{r}u = \mathbf{r}_1$$

for some elements a, u, b_1, $b_2 \in S$.

If $\mathbf{t_1}f_2 \sim \mathbf{t}_2$ and $\mathbf{t_2}f_1 \sim \mathbf{t}_1$ then $\mathbf{t_2} \sim \mathbf{t}_1$ in spite of our assumption. Therefore let us assume for instance that $\mathbf{t_1} \not\sim \mathbf{t}_2f_1$. (And so $\mathbf{t_1} \neq \mathbf{t}_2f_1$). This gives us an opportunity to consider $\mathbf{t}_2f_1$ instead of $\mathbf{t}_2$. So let us denote $\mathbf{t}_2 = \mathbf{t}_2f_1$, $f = f_1 = f_2$. Then $\mathbf{t}_2f = \mathbf{t}_2$, $\mathbf{t}_1f = \mathbf{t}_1$ and $\mathbf{t_1} \not\sim \mathbf{t}_2$. Now

$$\mathbf{p}eaf = \mathbf{p}af = \mathbf{r}$$
$$(\mathbf{p}, \mathbf{r}_1)eb_1f = (\mathbf{p}, \mathbf{r}_1)b_1f = (\mathbf{q}, \mathbf{t}_1)$$
$$(\mathbf{p}, \mathbf{r}_1)eb_2f = (\mathbf{p}, \mathbf{r}_1)b_2f = (\mathbf{q}, \mathbf{t}_2)$$
$$\mathbf{r}u = \mathbf{r}ue = \mathbf{r}_1$$

So we have



and $\mathbf{t_1} \not\sim \mathbf{t_2}$

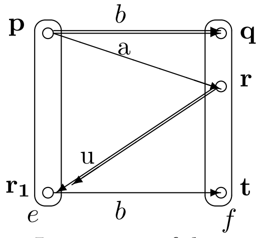
Let us denote $\mathbf{q}_1 = \mathbf{q}ue$ and $\mathbf{t} = \mathbf{q}_1 af_1$. Then
$(\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)e = (\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)$, $(\mathbf{q}, \mathbf{r})ue = (\mathbf{q}_1, \mathbf{r}_1)$, $(\mathbf{q}, \mathbf{r}, \mathbf{t}, \mathbf{t}_i)f = (\mathbf{q}, \mathbf{r}, \mathbf{t}, \mathbf{t}_i)$

So the node $(\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)$ is an *SCC*-node of the graph Γ^3, the nodes $(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{t}_i)$ are *SCC*-nodes of the graph Γ^4 for $i = 1, 2$ and we have $(\mathbf{q}, \mathbf{r}) \succeq (\mathbf{q}_1, \mathbf{r}_1)$, $(\mathbf{p}, \mathbf{q}_1) \succeq (\mathbf{r}, \mathbf{t})$ and $(\mathbf{p}, \mathbf{r}_1) \succeq (\mathbf{q}, \mathbf{t}_i)$ for $i = 1, 2$.

Therefore, by theorem 14, (2), we have $\mathbf{t}_1 = \mathbf{t}$ and $\mathbf{t}_2 = \mathbf{t}$. Hence, $\mathbf{t}_1 \sim \mathbf{t}_2$, contradiction. □

Definition 16 *For any four nodes* $\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{r}_1$ *of the graph* Γ *of a DFA such that* $\mathbf{p} \succeq \mathbf{r} \succeq \mathbf{r}_1$, $\mathbf{p} \succeq \mathbf{q}$ *and the nodes* $(\mathbf{p}, \mathbf{r}_1)$, $(\mathbf{q}, \mathbf{r})$ *are SCC-nodes, let* $T_{SCC}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{r}_1)$ *be the SCC of* Γ *containing*

$$T(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{r}_1) := \{t \mid (\mathbf{p}, \mathbf{r}_1) \succeq (\mathbf{q}, \mathbf{t}) \text{ and } (\mathbf{q}, \mathbf{r}, \mathbf{t}) \text{ is an } SCC\text{-node}\}$$



$\rightarrow \quad \mathbf{t} \in T_{SCC}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{r}_1)$

In virtue of lemma 15, the *SCC* $T_{SCC}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{r}_1)$ of a locally threshold testable *DFA* is well defined (but empty if the set $T(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{r}_1)$ is empty). Lemma 15 and theorem 14 (3) imply the following theorem

Theorem 17 *A DFA* $\mathbf{A}$ *with state transition graph* Γ *is locally threshold testable iff*

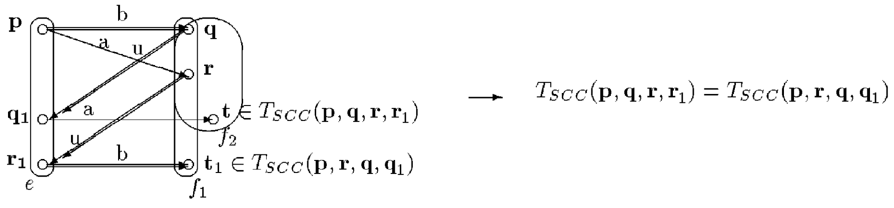
1)for every SCC-node $(\mathbf{p}, \mathbf{q})$ *of* Γ^2 $\mathbf{p} \sim \mathbf{q}$ *implies* $\mathbf{p} = \mathbf{q}$

and

2)for every five nodes $\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{q}_1, \mathbf{r}_1$ *of the graph* Γ *such that*

- *the non-empty SCC* $T_{SCC}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{r}_1)$ *and* $T_{SCC}(\mathbf{p}, \mathbf{r}, \mathbf{q}, \mathbf{q}_1)$ *exist,*
- *the node* $(\mathbf{p}, \mathbf{q}_1, \mathbf{r}_1)$ *is an SCC-node of the graph* Γ^3,
- $(\mathbf{q}, \mathbf{r}) \succeq (\mathbf{q}_1, \mathbf{r}_1)$ *in* Γ^2,

holds $T_{SCC}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{r}_1) = T_{SCC}(\mathbf{p}, \mathbf{r}, \mathbf{q}, \mathbf{q}_1)$.


p
q
r
q1
r1
t
t1
a
b
u
e
f1
f2
$\in T_{SCC}(\mathbf{p},\mathbf{q},\mathbf{r},\mathbf{r}_1)$
$\mathbf{t}_1 \in T_{SCC}(\mathbf{p},\mathbf{r},\mathbf{q},\mathbf{q}_1)$
$T_{SCC}(\mathbf{p},\mathbf{q},\mathbf{r},\mathbf{r}_1) = T_{SCC}(\mathbf{p},\mathbf{r},\mathbf{q},\mathbf{q}_1)$

2 Algorithm to Verify the Local Threshold Testability

A linear depth-first search algorithm finding all *SCC* of the given directed graph (see [1], [23] or [15]) will be used.

2.1 To Check the Reachability on an Oriented Graph

For a given node $\mathbf{q_0}$, we consider depth-first search from the node. First only $\mathbf{q_0}$ will be marked. Every edge is crossed two times. Given a node, the considered path includes first the ingoing edges and then the outgoing edges. After crossing an edge in the positive direction from the marked node $\mathbf{q}$ to the node $\mathbf{r}$ we mark $\mathbf{r}$ too. The process is linear in the number of edges (see [1], [13] for details).

The set of marked nodes forms a set of nodes that are reachable from $\mathbf{q_0}$. The procedure may be repeated for any node of the graph G.

The time of the algorithm for all pairs of nodes is $O(n^2)$.

2.2 To Verify Local Threshold Testability

Let us find all *SCC* of the graphs Γ, Γ^2 and Γ^3 and mark all *SCC*-nodes ($O(n^3)$ time complexity).

Let us recognize the reachability on the graph Γ and Γ^2 and form the table of reachability for all pairs of Γ and Γ^2. The time required for this step is $O(n^4)$.

Let us check the conditions of lemma 13. For every *SCC*-node $(\mathbf{p},\mathbf{q})$ $(\mathbf{p} \neq \mathbf{q})$ from Γ^2 let us check the condition $\mathbf{p} \sim \mathbf{q}$. A negative answer for any considered node $(\mathbf{p},\mathbf{q})$ implies the validity of the condition. In opposite case the automaton is not locally threshold testable. The time of the step is $O(n^2)$.

For every four nodes $\mathbf{p},\mathbf{q},\mathbf{r},\mathbf{r}_1$ of the graph Γ, let us check the following conditions (see 16): $\mathbf{p} \succeq \mathbf{r} \succeq \mathbf{r}_1$ and $\mathbf{p} \succeq \mathbf{q}$. In a positive case, let us form *SCC* $T_{SCC}(\mathbf{p},\mathbf{q},\mathbf{r},\mathbf{r}_1)$ of all nodes $\mathbf{t} \in \Gamma$ such that $(\mathbf{p},\mathbf{r}_1) \succeq (\mathbf{q},\mathbf{t})$ and $(\mathbf{q},\mathbf{r},\mathbf{t})$ is an *SCC*-node. In case that *SCC* T_{SCC} is not well defined the automaton is not threshold testable. The time required for this step is $O(n^5)$.

For every five nodes $\mathbf{p},\mathbf{q},\mathbf{r},\mathbf{q}_1,\mathbf{r}_1$ from Γ let us check now the second condition of theorem 17. If non-empty components $T_{SCC}(\mathbf{p},\mathbf{q},\mathbf{r},\mathbf{r}_1)$ and $T_{SCC}(\mathbf{p},\mathbf{r},\mathbf{q},\mathbf{q}_1)$ exist, the node $(\mathbf{p},\mathbf{q}_1,\mathbf{r}_1)$ is an *SCC*-node of the graph Γ^3 and $(\mathbf{q},\mathbf{r}) \succeq (\mathbf{q}_1,\mathbf{r}_1)$ in Γ^2, let us verify the equality $T_{SCC}(\mathbf{p},\mathbf{q},\mathbf{r},\mathbf{r}_1) = T_{SCC}(\mathbf{p},\mathbf{r},\mathbf{q},\mathbf{q}_1)$. If the answer is negative then the automaton is not threshold testable. A positive answer for all considered cases implies the validity of the condition of the theorem. The time required for this step is $O(n^5)$.

The whole time of the algorithm to check the local threshold testability is $O(n^5)$.

3 The Local Testability

We present now necessary and sufficient conditions of local testability of Kim, McNaughton and McCloskey ([13], [14]) in the following form:

Theorem 31 *([14]) A DFA with state transition graph Γ and transition semigroup S is locally testable iff the following two conditions hold:*

1)For any SCC-node $(\mathbf{p}, \mathbf{q})$ from Γ^2 such that $\mathbf{p} \sim \mathbf{q}$ we have $\mathbf{p} = \mathbf{q}$.

2)For any SCC-node $(\mathbf{p}, \mathbf{q})$ from Γ^2 such that $\mathbf{p} \succ \mathbf{q}$ and arbitrary element s from S we have $\mathbf{p}s \succeq \mathbf{q}$ is valid iff $\mathbf{q}s \succeq \mathbf{q}$.

The theorem implies

Corollary 32 *A DFA with state transition graph Γ over alphabet Σ is locally testable iff the following two conditions hold:*

1)For any SCC-node $(\mathbf{p}, \mathbf{q})$ from Γ^2 such that $\mathbf{p} \sim \mathbf{q}$ we have $\mathbf{p} = \mathbf{q}$.

2)For any node $(\mathbf{r}, \mathbf{s})$ and any SCC-node $(\mathbf{p}, \mathbf{q})$ from Γ^2 such that $(\mathbf{p}, \mathbf{q}) \succ (\mathbf{r}, \mathbf{s})$, $\mathbf{s} \sim \mathbf{q}$ and for arbitrary σ from Σ we have $\mathbf{r}\sigma \succeq \mathbf{s}$ is valid iff $\mathbf{s}\sigma \succeq \mathbf{s}$.

4 Algorithm to Verify the Local Testability

In [14], a polynomial time algorithm for local testability problem was considered. Now we present another version of such algorithm with the same time complexity. We hope that it will be more simple.

Let us form a table of reachability on the graph Γ ($O(n^2)$ time complexity).

Let us find Γ^2 and all SCC-nodes of Γ^2.

For every SCC-node $(\mathbf{p}, \mathbf{q})$ $(\mathbf{p} \neq \mathbf{q})$ from Γ^2 let us check the condition $\mathbf{p} \sim \mathbf{q}$. ($O(n^2)$ time complexity). If the condition holds then the automaton is not locally testable (32).

Let us exclude all edges $(\mathbf{p}, \mathbf{q}) \to (\mathbf{r}, \mathbf{s})$ from the graph Γ^2 such that $\mathbf{s} \not\succeq \mathbf{q}$ and $\mathbf{s} \not\succeq \mathbf{p}$. Then let us mark all nodes $(\mathbf{p}, \mathbf{q})$ of the graph Γ^2 such that for some σ from Σ from the two conditions $\mathbf{p}\sigma \succeq \mathbf{q}$ and $\mathbf{q}\sigma \succeq \mathbf{q}$ only one is valid. The time required for this step is $O(n^2)$.

Then we add to the graph new node $(\mathbf{0}, \mathbf{0})$ with edges from this node to every SCC-node. Let us find the set of nodes reachable from the node $(\mathbf{0}, \mathbf{0})$. ($O(n^2)$ time complexity). The automaton is locally testable iff no marked node belongs to obtained set (32).

The whole time of the algorithm to check the local testability is $O(n^2)$.

Acknowledgments. I would like to express my gratitude to Stuart Margolis for posing the problem and for helpful suggestions on improving the style of the paper and to referees for important and useful remarks.

References

1. A. Aho, J. Hopcroft, J. Ulman, The Design and Analisys of Computer Algorithms, Addison-Wesley, 1974.
2. J. Almeida, Implicit operations on finite J-trivial semigroups and a conjecture of I. Simon, J. Pure Appl. Alg., 69(1990), 205-208.
3. M.-P. Beal, J. Senellart, On the bound of the synchronization delay of local automata, Theoret. Comput. Sci. 205, 1-2(1998), 297-306.
4. D. Beauquier, J.E. Pin, Factors of words, Lect. Notes in Comp. Sci, Springer, Berlin, 372(1989), 63-79.
5. D. Beauquier, J.E. Pin, Languages and scanners, Theoret. Comp. Sci, 1, 84(1991), 3-21.
6. J.-C. Birget, Strict local testability of the finite control of two-way automata and of regular picture description languages, J. of Alg. Comp., 1, 2(1991), 161-175.
7. J.A. Brzozowski, I. Simon, Characterizations of locally testable events, Discrete Math., 4, (1973), 243-271.
8. P. Caron, LANGAGE: A Maple package for automaton characterization of regular languages, Springer, Lect. Notes in Comp. Sci., 1436(1998), 46-55.
9. Z. Esik, I. Simon, Modelling literal morphisms by shuffle. Semigroup Forum, 56(1998), 2, 225-227.
10. A. Ginzburg, About some properties of definite, reverse-definite and related automata, IEEE Trans. Electron. Comput. EC-15(1966), 806-810.
11. T. Head, Formal languages theory and DNA: an analysis of the generative capacity of specific recombinant behaviors, Bull. Math. Biol., 49(1987), 4, 739-757.
12. F. Hinz, Classes of picture languages that cannot be distinguished in the chain code concept and deletion of redundant retreats, Springer, Lect. Notes in Comp. Sci., 349(1990), 132-143.
13. S. Kim, R. McNaughton, R. McCloskey, An upper bound for the order of locally testable deterministic finite automaton, Lect. Notes in Comp. Sci., 401(1989), 48-65.
14. S. Kim, R. McNaughton, R. McCloskey, A polynomial time algorithm for the local testability problem of deterministic finite automata, IEEE Trans. Comput. 40(1991) N10, 1087-1093.
15. S. Kim, R. McNaughton, Computing the order of a locally testable automaton, Lect. Notes in Comp. Sci, Springer, 560(1991), 186-211.
16. G. Lallement, Semigroups and combinatorial applications, Wiley, N.Y., 1979.
17. S. W. Margolis, J.E. Pin, Languages and inverse semigroups, Lect. Notes in Comp. Sci, Springer, 199(1985), 285-299.
18. R. McNaughton, S. Papert, Counter-free automata, M.I.T. Press Mass., 1971.
19. M. Minsky, S. Papert, Perceptrons, M.I.T. Press Mass., 1971, Cambridge, MA, 1969.
20. J. Pin, Finite semigroups and recognizable languages. An introduction, Semigroups and formal languages, Math. and Ph. Sci., 466(1995), 1-32.
21. M. Perles, M. O. Rabin, E. Shamir, The theory of definite automata, IEEE Trans. Electron. Comput. EC-12(1963), 233-243.
22. J. Reiterman, The Birkhoff theorem for finite algebras, Algebra Universalis, 14(1982), 1-10.
23. R.E. Tarjan, Depth first search and linear graph algorithms, SIAM J. Comput., 1(1972), 146-160.

24. W. Thomas, Classifying regular events in symbolic logic, J. of Comp. System Sci, 25(1982), 360-376.
25. A.N. Trahtman, The varieties of n-testable semigroups, Semigroup Forum, 27(1983), 309-318.
26. A.N. Trahtman, A polynomial time algorithm for local testability and its level. Int. J. of Algebra and Comp., v. 9, 1(1998), 31-39.
27. A.N. Trahtman, The identities of k-testable semigroups. Comm. in algebra, v. 27, 11(1999), 5405-5412.
28. A.N. Trahtman, A precise estimation of the order of local testability of a deterministic finite automaton, Springer, Lect. Notes in Comp. Sci., 1436(1998), 198-212.
29. Th. Wilke, Locally threshold testable languages of infinite words, Lect. Notes in Comp. Sci, Springer, Berlin, 665(1993), 63-79.
30. Th. Wilke, An algebraic theory for regular languages of finite and infinite words, Int. J. Alg. and Comput. 3(1993), 4, 447-489.
31. Y. Zalcstein, Locally testable semigroups, Semigroup Forum, 5(1973), 216-227.
32. Y. Zalcstein, Locally testable languages, J. Comp. System Sci., 6(1972), 151-167.

A Taxonomy of Algorithms for Constructing Minimal Acyclic Deterministic Finite Automata

Bruce W. Watson

[1] University of Pretoria
(Department of Computer Science)
Pretoria 0002, South Africa
`watson@OpenFIRE.org`
`www.OpenFIRE.org`
[2] Ribbit Software Systems Inc.
(IST Technologies Research Group)

Abstract. In this paper, we present a taxonomy of algorithms for constructing minimal acyclic deterministic finite automata (MADFAs). Such automata represent finite languages and are therefore useful in applications such as storing words for spell-checking, computer and biological virus searching, text indexing and XML tag lookup. In such applications, the automata can grow extremely large (with more than 10^6 states) and are difficult to store without compression or minimization.
The taxonomization method arrives at all of the known algorithms, and some which are likely new ones (though proper attribution is not attempted, since the algorithms are usually of commercial value and some secrecy frequently surrounds the identities of the original authors).

1 Introduction

In this paper, we present a taxonomy of algorithms for constructing minimal acyclic deterministic finite automata (MADFAs). MADFAs represent finite languages and are therefore useful in applications such as storing words for spell-checking, computer and biological virus searching, text indexing and XML tag lookup. In such applications, the automata can grow extremely large and are difficult to store without compression or minimization. Whereas compression is considered in various other papers (and is usually specific to data-structure choices), here we focus on minimization.

We apply the following technique for taxonomizing the algorithms:

1. At the root of the taxonomy is a simple, if inefficient, algorithm whose correctness is either easy to prove or is simply postulated.
2. New algorithms are derived by adding an algorithm detail — a correctness-preserving transformation of the algorithm or elaboration of program statements. This yields an algorithm which is still correct.
3. By carefully choosing the details, all of the well-known algorithms appear in the taxonomy. Creative invention of new details also yields new algorithms.

This technique was applied on a large scale in the my Ph.D dissertation [5]. The dissertation also contains taxonomies of algorithms for constructing finite automata from regular expressions and for minimizing deterministic finite automata. Here, we assume some familiarity with the common algorithms for automata construction and minimization.

1.1 Related Work

The work presented here is significantly different from the taxonomies presented in the dissertation, since specializing for MADFAs can yield particularly efficient algorithms.

Some of the algorithms included in this taxonomy were previously presented, for example, in Turkey [1] (the present author, Jan Daciuk and Richard Watson), at WIA'98 [6] (the present author) and in [3] (Stoyan Mihov). Other algorithms for the MADFA construction problem have typically been kept as trade secrets (due to their commercial success in applications such as spell-checking). As such, many of them have likely been known for some number of years, but tracing the original authors will be difficult and proper attributions are not attempted — though I would welcome hearing from researchers who performed some of the original work.

1.2 Preliminaries

We make the following definitions:

– FA is the set of all *finite automata*.
– DFA is the set of all *deterministic* FAs.
– ADFA is the set of all *acyclic* DFAs.
– MADFA is the set of all *minimal* ADFAs.

More precise definitions are not required here. In this paper, we are primarily interested in algorithms which build MADFAs. The algorithms are readily extended to work with acyclic deterministic *transducers*, though such an extension is not considered.

For any $M \in \mathsf{FA}$, $|M|$ is the number of states in M and $\mathcal{L}(M)$ is the language (set of words) accepted by M. The primary definition of minimality of an $M \in \mathsf{DFA}$ is: $(\forall\ M' \in \mathsf{DFA} : \mathcal{L}(M') = \mathcal{L}(M) : |M| \leq |M'|)$.

Predicate $Min(M)$ holds when $M \in \mathsf{DFA}$ and the above definition of minimality both hold. A useful DFA property is: $\mathcal{L}(M)$ is finite $\wedge\ Min(M) \equiv M \in \mathsf{MADFA}$.

All of the algorithms presented here are in the guarded command language, a type of pseudo-code — see [2].

2 A First Algorithm

In this section, we present our first algorithm and outline some ways in which to proceed. The problem is as follows: given alphabet Γ and some finite set of words $W \subset \Gamma^*$ (the containment is proper, since Γ^* is infinite), compute some $M \in \mathsf{ADFA}$ such that $\mathcal{L}(M) = W \wedge Min(M)$. In the algorithms that follow, we give M the type FA, which is the most general type in the containment $\mathsf{MADFA} \subset \mathsf{ADFA} \subset \mathsf{DFA} \subset FA$. At any point in the program, the variable M may actually contain a MADFA.

Given this, our first algorithm (where S is a program statement still to be derived) is:

Algorithm 2.1:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

□

In order to make some progress, we consider a split of statement S to accomplish the postcondition in two steps:

Algorithm 2.2:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0$;
{ $\mathcal{L}(M) = f(W) \wedge X(M)$ }
$S_1$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

□

There are, of course, infinitely many choices for function f and predicate X, some of which are not interesting. For example, if we define $f(W) = \emptyset$, then after S_0, we will have accomplished virtually nothing (since the automaton will accept the empty language), regardless of how we define X. For this reason, we restrict ourselves to the following three possibilities for f:

1. $f(W) = W$ (the identity function).
2. $f(W) = W^R$ (the reversal of the W).
3. $f(W) = \neg W$ (the complement of W: $\neg W = \Gamma^* - W$).

Other choices are possible. These were chosen because:

– in some sense, statement S_0 will accomplish a reasonable amount of work,
– it is reasonably easy to convert an $f(W)$-accepting DFA to a W-accepting one, and
– with these choices, we can arrive at many of the known algorithms.

We consider each choice of f in the following sections.

3 $f(W) = W$

We can now turn to choices for predicate X. Clearly, any predicate can be chosen, but we restrict our choices to (at least) arrive at the known algorithms. As a first option, consider *strengthenings* of *Min*, that is: $X(M) \Rightarrow \mathit{Min}(M)$. In that case, we choose S_1 to be the **skip** statement (which does nothing, since $\mathit{Min}(M)$ already holds), and we are left with a statement S_0 which is as difficult to derive as our first algorithm. For this reason, we abandon strengthenings of *Min* (including the possiblity $X(M) \equiv \mathit{Min}(M)$).

Instead, we turn our attention to weakenings[1] of *Min*. We begin with the extreme of these weakenings: *true*.

3.1 $X(M) \equiv \mathit{true}$

By writing our choices of f and X in full, our program becomes:

Algorithm 3.1:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0$;
{ $\mathcal{L}(M) = W$ }
$S_1$
{ $\mathcal{L}(M) = W \wedge \mathit{Min}(M)$ }
```

□

For S_0, we can use any algorithm which yields an automaton M such that $\mathcal{L}(M) = W$. In §7, we separately consider algorithms for doing this.

If the expansion of S_0 is an algorithm yielding a DFA, then for S_1 we can use any of the minimization algorithms in [5, Chapter 7] or the one given by [4]. If M delivered by S_0 is not deterministic, we can either use Brzozowski's minimization algorithm (see [5, Chapter 7]) or first apply the subset construction (to determinize M) and then any one of the other minimization algorithms.

Clearly the extensive choices for S_0 and S_1 yield an entire subtree of the taxonomy — and therefore an entire family of algorithms.

3.2 $X(M) \equiv M \in \mathsf{DFA}$

This yields:

[1] We could equally choose some X which is not related by implication to *Min*; this has not been explored and is a topic for future research, since it may lead to interesting algorithms.

Algorithm 3.2:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0$;
{ $\mathcal{L}(M) = W \wedge M \in \mathsf{DFA}$ }
$S_1$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

□

The choices for S_0 are as in §3.1 and are discussed in §7. Similarly, for S_1, there are a number of choices (see [5, Chapter 7] and [4]) — though we are already certain that M is a DFA and no determinization step is required.

3.3 $X(M)$ as Partial Minimality

In [6], a partial minimality predicate is introduced and it is shown to be a weakening of *Min*. This yields the following algorithm:

Algorithm 3.3:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0$;
{ $\mathcal{L}(M) = W \wedge X(M)$ }
$S_1$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

□

In the original paper, S_0 is derived as an algorithm which constructs M as a *partially minimal* DFA, while S_1 is derived as a 'cleanup' phase to finalize the minimization. The interested reader is referred to the presentation in that paper.

4 $f(W) = W^R$

It is no accident that reversal was used in f: it is known to be related to minimality via Brzozowski's minimization algorithm [5] (in that presentation, the history of the algorithm is given, along with full correctness arguments for each part of the algorithm). Brzozowski's algorithm, for some $M \in \mathsf{FA}$ (not necessarily a DFA), is:

Algorithm 4.1:

```
$M' := reverse(M)$;
$M' := determinize(M')$;
{ $\mathcal{L}(M') = \mathcal{L}(M)^R \wedge M' \in \mathsf{DFA}$ }
$M' := reverse(M')$;
$M' := determinize(M')$
{ $\mathcal{L}(M') = \mathcal{L}(M) \wedge Min(M')$ }
```

□

Thanks to this, the most obvious choice for predicate X is $X(M) \equiv M \in \mathsf{DFA}$. In that case, our program is

Algorithm 4.2:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0$;
{ $\mathcal{L}(M) = W^R \wedge M \in \mathsf{DFA}$ }
$S_1$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

$\square$

Using Brzozowski's algorithm, we expand S_1 in the above program:

Algorithm 4.3:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0$;
{ $\mathcal{L}(M) = W^R \wedge M \in \mathsf{DFA}$ }
$M := reverse(M)$;
$M := determinize(M)$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

$\square$

For S_0, there are a number of algorithms for building a DFA from W (see §7), and we can trivially modify them to deal with W^R.

5 $f(W) = \neg W$

It is known that DFA minimality is preserved under negation of the DFA, at least using most reasonable definitions of a negating mapping[2]. Armed with this, we choose $X(M) \equiv Min(M)$. This yields

Algorithm 5.1:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0$;
{ $\mathcal{L}(M) = \neg W \wedge Min(M)$ }
$S_1$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

$\square$

[2] We assume a negating mapping which is able to work on DFAs with partial transition functions, since a total transition function would (in the case of non-empty DFAs) be cyclic.

With the preservation of minimality under negation, we select S_1 to be the negation function, giving

Algorithm 5.2:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0$;
{ $\mathcal{L}(M) = \neg W \wedge Min(M)$ }
$M := negate(M)$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

□

Statement S_0 can be further split, giving

Algorithm 5.3:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$S_0'$;
{ $\mathcal{L}(M) = \neg W$ }
$S_0''$;
{ $\mathcal{L}(M) = \neg W \wedge Min(M)$ }
$M := negate(M)$
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

□

In this case, S_0' builds M corresponding to $\neg W$; this can be accomplished by first building M corresponding to W and then applying *negate*. (There may, of course, be other algorithms still to be derived.) Subsequently, S_0'' corresponds to some minimization algorithm, for example, those given in [4,5]. The running time advantages of including this negation step are not yet clear.

6 *Min*(*M*) as an Invariant

In the previous section, we have considered algorithms with two parts: S_0 and S_1. We return to Algorithm 2.1 — the root of the taxonomy — to obtain the following algorithm, where we use $Min(M)$ as a repetition invariant:

Algorithm 6.1:

```
{ $W \subseteq \Gamma^* \wedge W$ is finite }
$M := empty\_DFA$;
$Done, To\_do := \emptyset, W$;
{ invariant: $\mathcal{L}(M) = Done \wedge Min(M) \wedge Done \cup To\_do = W \wedge Done \cap To\_do = \emptyset$
  variant: $|To\_do|$ }
do $To\_do \neq \emptyset \rightarrow$
    $S_2$; { choose some word in $To\_do$ }
    { $w \in To\_do$ }
    $Done, To\_do := Done \cup \{w\}, To\_do - \{w\}$;
    $S_3$
od
{ $Done = W$ }
{ $\mathcal{L}(M) = W \wedge Min(M)$ }
```

□

We now consider possible versions of statements S_2 and S_3. There are two straightforward ways to proceed with S_2:

1. *Lexicographically order the words in W.* Obtaining the elements of W in lexicographic order is easily implemented. To implement statement S_3, a derivation was recently given in [1], and the interested reader is referred to that paper.
2. *Unordered choice from W.* This is the easiest way in which to select an element of W. As above, an implementation of S_3 was also derived in [1], and it is not considered in detail here.

These two algorithms are the only two fully incremental MADFA construction algorithms known. Both of them have running time which is linear in the size of W (as does Revuz's algorithm [4] — an algorithm related to the two mentioned here).

7 Constructing a (Not Necessarily Minimal) Finite Automaton

In this section, we briefly discuss some algorithms for constructing a finite automaton from W:

1. One obvious (though not very efficient) method is to first build a regular expression from W (as $w_0 + w_1 + \cdots + w_{|W|-1}$ for words $w_i \in W$) and then use one of the general construction algorithms given in [5, Chapter 6]. This algorithm has not yet been benchmarked, although it is likely to be slow due to the generality. It is possible, however, that some improvements could be made based upon the simple (star-free) structure of the regular expressions.
2. For each $w \in W$, we build a simple linear finite automaton with $|w| + 1$ states (the transitions are respectively labeled with the letters from w). The final (nondeterministic) finite automaton is built by combining all of the individual automata and adding a new start state with ε-transitions to the individual start states. As with the above algorithm, this one is very likely to be slow.
3. For each $w \in W$, we apply the standard algorithm for adding a word to a *trie*-structured DFA. Such algorithms are presented in most algorithm texts.

8 Conclusions

We have presented a straightforward taxonomy of algorithms for constructing minimal acyclic deterministic finite automata. The taxonomy begins with an algorithm which has unelaborated statements, postulated to be correct. Each of the subsequent algorithms is derived by applying correctness-preserving transformations to the initial algorithm. In the course of constructing the taxonomy, all of the pre-existing algorithms were derived — including some of the most recently presented incremental algorithms. Furthermore, the taxonomy elaborated on two other groups of algorithms:

– Many of the original, and efficient, algorithms were previously only known as trade secrets in industry.
– Some of the intermediate algorithms contain dead-ends or have derivation possibilities which are unexplored.

There are a number of areas of future research:

– Although most of the algorithm details are intuitively correct, the full correctness arguments must be provided.
– A number of unexplored directions were highlighted in the taxonomy. Some of these may, in fact, lead to new algorithms of practical importance.
– The theoretical and benchmarked running time of the algorithms has not been adequately explored and are not given in this paper. This will allow the careful choice of an algorithm to apply in practice.

Acknowledgements. I would like to thank Nanette Saes and the anonymous referees for improving the quality of this paper.

References

1. Daciuk, J.D., Watson, B.W. and R.E. Watson. An Incremental Algorithm for Constructing Acyclic Deterministic Transducers. (Proceedings of the International Workshop on Finite State Methods in Natural Language Processing, Ankara, Turkey, 30 June–1 July 1998).
2. Dijkstra, E.W. *A Discipline of Programming.* (Prentice Hall, Englewood Cliffs, N.J., 1976).
3. Mihov, S. Direct Building of Minimal Automaton for Given List. (Available from `stoyan@lml.acad.bg`).
4. Revuz, D. Minimisation of acyclic deterministic automata in linear time. (Theoretical Computer Science 92, pp. 181-189, 1992).
5. Watson, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* (Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995). See `www.OpenFIRE.org`
6. Watson, B.W. A Fast New Semi-Incremental Algorithm for the Construction of Minimal Acyclic DFAs. (Proceedings of the Third Workshop on Implementing Automata, Rouen, France, September 1998).

Author Index