Michael Domaratzki
Kai Salomaa (Eds.)

Implementation and Application of Automata

LNCS 6482

15th International Conference, CIAA 2010
Winnipeg, MB, Canada, August 2010
Revised Selected Papers

Springer

Lecture Notes in Computer Science 6482

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Michael Domaratzki Kai Salomaa (Eds.)

Implementation and Application of Automata

15th International Conference, CIAA 2010
Winnipeg, MB, Canada, August 12-15, 2010
Revised Selected Papers

Springer

Volume Editors

Michael Domaratzki
University of Manitoba, Department of Computer Science
Winnipeg, MB, R3T 2T2, Canada
E-mail: mdomarat@cs.umanitoba.ca

Kai Salomaa
Queen's University, School of Computing
Kingston, ON, K7L 3N6, Canada
E-mail: ksalomaa@cs.queensu.ca

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-18097-2 e-ISBN 978-3-642-18098-9
DOI 10.1007/978-3-642-18098-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2010942794

CR Subject Classification (1998): F.2, F.1, G.2, F.3, E.1, F.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume of *Lecture Notes in Computer Science* contains revised versions of papers presented at the 15th International Conference on Implementation and Application of Automata, CIAA 2010. The conference was held at the University of Manitoba in Winnipeg, Canada, on August 12–15, 2010. The previous CIAA conferences were held in London, Ontario (2000), Pretoria (2001), Tours (2002), Santa Barbara (2003), Kingston (2004), Nice (2005), Taipei (2006), Prague (2007), San Francisco (2008) and Sydney (2009).

The CIAA meeting can be viewed as the main conference for researchers, application developers, and users of automata-based systems. The topics of the conference include applications of automata in, for example, computer-aided verification, natural language processing, pattern matching, data storage and retrieval, and bioinformatics, as well as foundational work on automata theory.

The 26 full papers and 6 short papers were selected from 52 submissions. Each submitted paper was evaluated by at least three Program Committee members, with the help of external referees. We warmly thank the invited speakers, the authors of the contributed papers, as well as the reviewers and the Program Committee members for their valuable work. All these efforts were the basis for the success of the conference.

During the conference, Cyril Allauzen, with co-authors Corinna Cortes and Mehryar Mohri, was presented with the CIAA 2010 Best Paper award for their paper entitled "Large-Scale Training of SVMs with Automata Kernels." The paper describes transducer-based methods for improving the efficiency of training SVMs in machine-learning applications.

The authors of the papers included in these proceedings come from the following countries: Canada, China, Czech Republic, France, Germany, Hungary, Italy, Japan, Poland, Portugal, Spain, Taiwan, the UK and the USA. In addition, the conference had participants with affiliations in Belgium and Finland.

We thank our sponsors for their generous financial support: the Fields Institute; MITACS; Office of the Vice-President (Research), University of Manitoba; and the Department of Computer Science, University of Manitoba. To conclude, we are indebted to Alfred Hofmann and Anna Kramer from Springer for the efficient collaboration in producing this volume.

October 2010

M. Domaratzki
K. Salomaa

Organization

Invited Speakers

Natasha Jonoska	University of South Florida, USA
Madhusudan Parthasarathy	University of Illinois at Urbana-Champaign, USA
Karen Rudie	Queen's University, Canada

Program Committee

Marie-Pierre Béal	Université de Marne-la-Vallée, France
Cezar Câmpeanu	University of Prince Edward Island, Canada
Pascal Caron	Université de Rouen, France
Jean-Marc Champarnaud	Université de Rouen, France
Mark Daley	University of Western Ontario, Canada
Michael Domaratzki (Chair)	University of Manitoba, Canada
Yo-Sub Han	Yonsei University, South Korea
Tero Harju	University of Turku, Finland
Markus Holzer	Technische Universität München, Germany
Oscar Ibarra	University of California, Santa Barbara, USA
Lucian Ilie	University of Western Ontario, Canada
Masami Ito	Kyoto Sangyo University, Japan
Stavros Konstantinidis	Saint Mary's University, Canada
Igor Litovsky	Université de Nice, France
Carlos Martín-Vide	Rovira i Virgili University, Spain
Sebastian Maneth	NICTA; University of New South Wales, Australia
Denis Maurel	Université de Tours, France
Ian McQuillan	University of Saskatchewan, Canada
Mehryar Mohri	Courant Institute of Mathematical Sciences, USA
Alexander Okhotin	University of Turku, Finland
Andrei Păun	Louisiana Tech University, USA; University of Bucharest, Romania
Giovanni Pighizzini	Università degli Studi di Milano, Italy
Bala Ravikumar	Sonoma State University, USA
Rogério Reis	Universidade do Porto, Portugal
Kai Salomaa (Co-chair)	Queen's University, Canada
Bruce Watson	University of Pretoria, South Africa; Sagantec, USA
Hsu-Chun Yen	National Taiwan University, Taiwan

Sheng Yu	University of Western Ontario, Canada
Djelloul Ziadi	Université de Rouen, France

Additional Referees

Cyril Allauzen	Dora Giammarresi	Nelma Moreira
Jean-Paul Allouche	Sebastian Jakobi	Benedek Nagy
Marco Almeida	Artur Jez	Florent Nicart
Nicolas Bedon	Derrick Kourie	Xiaoxue Piao
Jean-Camille Birget	Martin Kutrib	Narad Rampersad
Sabine Broda	Eric Laugerotte	Michael Riley
Bernd Burgstaller	Tommi Lehtinen	Agata Savary
Michael Burrell	Aurelien Lemay	Shinnosuke Seki
Olivier Carton	Peter Leupold	Tinus Strauss
Julien Cervelle	Beth Locke	Jean-Yves Thibon
Alfredo Costa	Sylvain Lombardy	Mauro Torelli
Eugen Czeizler	Violetta Lonati	Nicholas Tran
Pal Domosi	António Machiavelo	Andrea Visconti
Nathalie Friburger	Andreas Malcher	Mikhail Volkov
Zoltan Fulop	Andreas Maletti	
Christopher Geib	Yoshihiro Mizoguchi	

Sponsors

Fields Institute

MITACS

University of Manitoba

Table of Contents

Using Automata to Describe Self-Assembled Nanostructures

Nataša Jonoska*

Department of Mathematics
University of South Florida, Tampa, FL 33620, USA
jonoska@math.usf.edu

There is an increased necessity for mathematical study of self-assembly of various phenomena ranging from nano-scale structures, material design, crystals, biomolecular cages such as viral capsids and for computing. We show an algebraic model for describing and characterizing nanostructures built by a set of molecular building blocks. This algebraic approach connects the classifcal view of crystal dissection with a more modern system based on algebraic automata theory.

A molecular building block is represented as a star-like graph with a central vertex and molecular bonding sites at the single-valent vertices. Such a graph represents a branched junction molecule whose arms contain "sticky ends" ready to connect with other molecular blocks. The building blocks have specific chemical properties on their bonding sites presented as labels or colors. These bonds may be strong covalent or weak ionic (hydrogen) types of bonds and specify which two sites can be superimposed or "glued" together. The connection or bonding is allowed only along "compatible" sites, where the "compatibility" is defined by a binary relation on the set of bond types. One can consider two types of molecular building blocks, flexible and rigid. In the case of flexible building blocks, the assembly of blocks in larger structures is guided only by the binary relation specifying the bonding, without any geometric constraints. In the case of rigid molecular building blocks, the geometry of the molecular building block plays a major role in the assembly process.

We concentrate on two general problems: (1) how to characterize or classify structures that can be built? (2) how can two non-congruent structures be distinguished?

For these questions we propose to use a finite state automaton whose states are the molecular building block types and the transitions of the automaton are the bonding operations. In the case of flexible building blocks (which we call flexible tiles) the bonding operation is just the binary operation that specifies the bonds, and therefore one can use the bond types as transition labels.

In the case of rigid building blocks (or rigid tiles) one utilizes classical symmetry isometries based on translation and rotation as transition labels. If a block is used in an assembly of a complex structure, it needs to be displaced by a vector $\mathbf{x}$ and then rotated by an angle θ about a line ℓ in order to meet the appropriate bonding

* This work has been supported in part by the NSF grants CCF-0726396 and DMS-0900671. Based on joint work with G.L. McColm.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 1–3, 2011.

sites. Its new placement can be described by the triple $(\mathbf{x}, \ell, \theta)$. All block types that take part in an assembly of the structure are assumed to be taken from a standard position (centered at the origin in a particular orientation) and then displaced to their appropriate position in the built structure. For every assembled structure we assume that there is a building block which is in its standard position which we call a *reference block*.

We also assume that there is a finite set of building block types, there are a finite number of bond types, and a finite number of relative rotations that one block can take with respect to another while bonding.

A structure can be described by knowing the positions of all building blocks, and those positions are known if the paths from the reference block to each block in the structure are known. Hence, we can describe a structure (composed of building blocks) by describing all of the paths that traverse it. Every path starts at the center of the reference block, and moves from block to block across bonds.

A finite state automaton can be used to generate the possible paths for a given set of block types within an assembled structure. The states of the automaton are the block types. The automaton also has a zero state 0 to represent a state of an illegal move. All states of the automaton are initial and terminal. The transition alphabet of the automaton consists of all possible movements from one block (of a specific type) to another block (of a specific type). As mentioned, in the case of flexible tiles, the transition alphabet is just the set of bonding types. In the case of rigid tiles, a transition is specified by a rotation and displacement vector in addition to the bonding type. For example, a transition from a block $\mathbf{b}$ to a block $\mathbf{b}'$ labeled by symbol $\sigma = (\mathbf{u}, \imath, \varphi)$ represents the following movement. From a block $\mathbf{b}$, rotate through an angle φ about a line $\imath$ (for $\imath$ displaced to go through the center of $\mathbf{b}$) and translate through the vector $\mathbf{u}$ to a new block $\mathbf{b}'$.

To each walk within the structure we associate a walk in the directed graph represented by the finite state automaton. A framed walk is a list of instructions for walking through a structure. Intuitively we take that "structure" is a possible assembled structure that can be obtained by gluing building blocks of given types according to the bonding relation. We have the following definition.

Definition 1. *Let $\mathfrak{C}$ be a structure, and let $\mathbf{b}$ be a block of $\mathfrak{C}$. Let $W(\mathfrak{C}, \mathbf{b})$ be the set of framed walks in the automaton that can be walked in $\mathfrak{C}$ starting at $\mathbf{b}$ as a reference block. Furthermore, the* length *of a framed walk is the number of transitions, i.e., $|\mathfrak{b}\sigma_1\sigma_2\cdots\sigma_n\mathfrak{b}'| = n$. Let $W_n(\mathfrak{C}, \mathbf{b}) = \{w \in W(\mathfrak{C}, \mathbf{b}) \colon |w| \leq n\}$.*

We now convert geometry to algebra by defining an equivalence that identifies pairs of framed walks that lead from the same initial block to the same terminal block. Let $\mathfrak{b}_0\mathbf{s}_1\mathfrak{b} \sim \mathfrak{b}_0\mathbf{s}_2\mathfrak{b}$ mean that if we start at a block of type $\mathfrak{b}_0$ (in standard position) and follow the walk instructions $\mathbf{s}_1$, we would wind up in the same position and orientation, *and hence the same block* as if it followed the walk instructions $\mathbf{s}_2$. We define a structure to be *rigid with respect to* $\sim$ if for any three blocks $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2$ of the structure, where $\text{type}(\mathbf{b}_0) = \mathfrak{b}_0$ and $\text{type}(\mathbf{b}_1) = \text{type}(\mathbf{b}_2) = \mathfrak{b}$, the following is true: $\mathbf{b}_1 = \mathbf{b}_2$ if and only if for any framed walk $\mathfrak{b}_0\mathbf{s}_1\mathfrak{b}$ from $\mathbf{b}_0$ to $\mathbf{b}_1$ and for any framed walk $\mathfrak{b}_0\mathbf{s}_2\mathfrak{b}$ from $\mathbf{b}_0$ to $\mathbf{b}_2$, $\mathfrak{b}_0\mathbf{s}_1\mathfrak{b} \sim \mathfrak{b}_0\mathbf{s}_2\mathfrak{b}$.

Notice that $\sim$ depends on the shapes of the blocks, and on the geometric space in which the blocks are placed. This relation $\sim$ is defined through cycles in the walkspace, i.e., by identifying those framed walks that in the geometry they lead back to the block on which they started. For a set of walks W denote with $\hat{W}$ the equivalence classes of walks in W.

In the case of flexible tiles, $\sim$ becomes trivial, since the flexibility of the building blocs assures that there are no constraints for bonding of the tiles as soon as there are available (non-attached) bonds.

This notion of rigidity of the structure has the following consequence.

Theorem 1. *Suppose that two structures* $\mathfrak{C}_1$ *and* $\mathfrak{C}_2$ *which consist of* n *blocks each are generated from the same block types according to transitions of the same automaton. Then they are (geometrically) congruent if and only if for some block* $\mathbf{b}_1$ *of* $\mathfrak{C}_1$ *and some block* $\mathbf{b}_2$ *of* $\mathfrak{C}_2$, $\hat{W}_n(\mathfrak{C}_1, \mathbf{b}_1) = \hat{W}_n(\mathfrak{C}_2, \mathbf{b}_2)$.

Hence we can treat a structure as a set of framed walks, i.e., as a language, thus removing the geometry from the computation. Using this fact, and *assuming that* $\sim$ *(and associated apparatus) is PTIME computable*, we obtain that congruence of rigid structurees is PTIME computable, which contrasts from the popular suspicion that the Graph Isomorphism problem is *not* PTIME computable.

A Summary of Some Discrete-Event System Control Problems*

Karen Rudie

Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario K7L 3N6
Canada

Abstract. A summary of the area of control of discrete-event systems is given. In this research area, automata and formal language theory is used as a tool to model physical problems that arise in technological and industrial systems. The key ingredients to discrete-event control problems are a process that can be modeled by an automaton, events in that process that cannot be disabled or prevented from occurring, and a controlling agent that manipulates the events that *can* be disabled to guarantee that the process under control either generates all the strings in some prescribed language or as many strings as possible in some prescribed language. When multiple controlling agents act on a process, *decentralized* control problems arise. In decentralized discrete-event systems, it is presumed that the agents effecting control cannot each see all event occurrences. Partial observation leads to some problems that cannot be solved in polynomial time and some others that are not even decidable.

Keywords: Discrete-Event Systems, Supervisory Control, Decentralized Control.

1 Introduction

Discrete-event systems are processes whose behaviour can be characterized by sequences of events. Behaviour of the system is captured by a formal language and is typically represented by an automaton that recognizes that language. It is assumed that some events that the system can generate can be prevented from occurring (by an external agent, which may be software, hardware or a human operator) and some other events cannot be prevented from occurring. Those events that can be disabled are called *controllable* events and those that cannot be disabled are called *uncontrollable* events. Control problems arise because the systems can generate undesirable event sequences. Work in this area typically addresses when it is possible to derive agents that can prohibit bad sequences by disabling controllable events at various points along the strings that can be

* This work was supported by a Discovery Grant from the Natural Sciences and Engineering Research Council (NSERC).

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 4–16, 2011.

generated. These problems are more difficult computationally if they must be solved using decentralized control, where there are multiple agents, each of which has only a partial view of overall system behaviour and has only partial control.

The research area of *supervisory control* of discrete-event systems takes a control-theoretic approach to solving problems that arise when dealing with discrete-event processes but the technical machinery employed is all based on automata theory and the theory of formal languages.

2 Supervisory Control Problems

The work described in this section is set in the supervisory control framework for discrete-event systems developed by P.J. Ramadge and W.M. Wonham in the early 1980s and initiated by the doctoral thesis of Ramadge [1, 2]. For more details, refer to the primary textbook in discrete-event control systems [3].

The essence of discrete-event system (DES) control problems is the following. There is a process, called the *plant* G, whose behaviour can be thought of as sequences of events or actions. Some of the event sequences in G are undesirable (e.g., perhaps because they contravene some safety specifications) and some of the events in G can be prevented from occurring (i.e., "disabled"). The event sequences that are not undesirable are called the *legal* sequences. An agent, called a *supervisor*, is responsible for controlling G to ensure that the legal sequences occur, or if that's not possible, that only some subset of legal sequences occur. By "agent" we mean a computer program or hardware or a human operator or any entity that would be able to disable certain events and that would have some means of observing (or sensing) at least some of the event occurrences.

Because plants are nothing more than generators of sequences of discrete events, they can be modelled using automata. Consequently, the plant G is given by an automaton over the alphabet Σ:

$$G = (Q, \Sigma, \delta, q_0, Q_m) \ .$$

Although most of the examples in DES literature involve finite automata, the general setting and theoretical results do not presume at the outset that the state set is finite. When the state set *is* finite, we often visually represent the automaton with a state-transition diagram. For example, consider the automaton depicted in Figure 1. This plant might be describing part of a telecommunications network, where some process sends data, the data is then either received or lost, and if it is lost, it is re-sent. In DES parlance, we refer to terminal states as *marked states* because we use those states to demarcate certain strings (generated by the plant) for special consideration. Typically, sequences that end at marked states would distinguish completed tasks. For example, in Figure 1, by marking only state 4, we are able to speak about the sequences describing scenarios where the data sent is actually received.

In the context of supervisory control of discrete-event systems, automata are used as a way of representing event sequences. The set of all possible event sequences is given by Σ^*. The transition function of G can be extended from

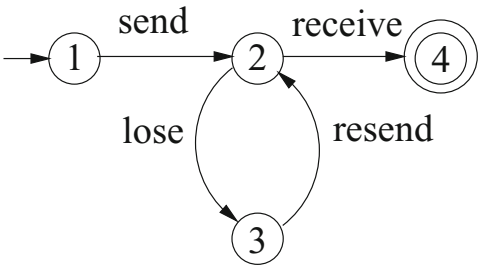


Fig. 1. State-Transition Diagram for Part of a Telecommunications Network

single event occurrences to event sequences: the idea is that for some $s \in \Sigma^*$, $\delta(q_0, s)$ indicates the state to which the *sequence* of events in s leads. We associate with an automaton G two languages defined as follows:

$$L(G) := \{s \mid s \in \Sigma^* \text{ and } \delta(q_0, s) \text{ is defined}\}$$

and

$$L_m(G) := \{s \mid s \in \Sigma^* \text{ and } \delta(q_0, s) \in Q_m\} \ .$$

The language $L(G)$ is the set of all possible event sequences which the plant may generate (and is sometimes called the *closed behaviour*). The language $L_m(G)$ is intended to distinguish some subset of possible plant behaviour as representing completed tasks (and is sometimes called the *marked behaviour*). Note the departure here from both the terminology and notation of standard automata theory: the set $L(G)$ is *not* the set of strings accepted by the automaton G; in the DES community we use the notation $L_m(G)$ for the set of strings accepted by G. In contrast, the set $L(G)$ is all those strings that the transition function δ admits; they need not lead to a marked (i.e., terminal) state of G.

To impose supervision on the plant, we identify some of its events as *controllable* and some as *uncontrollable*, thereby partitioning Σ into the disjoint sets Σ_c, the set of controllable events, and Σ_{uc}, the set of uncontrollable events. Controllable events are those which an external agent may enable (permit to occur) or disable (prevent from occurring) while uncontrollable events are those which cannot be prevented from occurring and are therefore considered to be permanently enabled. For the plant in Figure 1 a reasonable partition of events would be that "send" and "resend" are controllable events, whereas "lose" and "receive" are uncontrollable (since once a message is sent, one cannot prevent it from being lost). A *supervisor* (sometimes called a *controller*) is then an agent which observes a sequence of events as it is generated by G and enables or disables any of the events under its control at any point in time throughout its observation. By performing such a manipulation of controllable events, the supervisor ensures that only a subset of $L(G)$ is permitted to be generated (and this subset captures those sequences that are "desirable" in some way). To capture the notion that a supervisor may only see *some* of the events generated

by the plant, the event set Σ is also partitioned into the disjoint sets Σ_o of observable events and Σ_{uo} of unobservable events.

To model a supervisor's partial view of the plant, we use a mapping called the *projection*, which we interpret as a supervisor's view of the strings in Σ^*. The projection $P : \Sigma^* \longrightarrow \Sigma_o^*$ is defined recursively as follows:

$$P(\varepsilon) := \varepsilon,$$

for $\sigma \in \Sigma$,

$$P(\sigma) := \begin{cases} \sigma \text{ if } \sigma \in \Sigma_o \\ \varepsilon \text{ otherwise} \end{cases}$$

and for $s \in \Sigma^*, \sigma \in \Sigma$,

$$P(s\sigma) := P(s)P(\sigma).$$

In other words, P erases all events that are unobservable to the supervisor.

Formally, a *supervisor* $\mathcal{S}$ is a pair (T, ψ) where T is an automaton which recognizes a language over the same event set as the plant G and ψ, called a *feedback map*, is a map from Σ and states of T to the set $\{enable, disable\}$. If X denotes the set of states of T, then $\psi : \Sigma \times X \to \{enable, disable\}$ satisfies

$$\psi(\sigma, x) = enable \text{ if } \sigma \notin \Sigma_c,\ x \in X,$$

i.e., a supervisor cannot disable an uncontrollable event. In addition, $\mathcal{S}$ must satisfy the condition that if an event σ is unobservable to the supervisor, then the supervisor cannot change state upon the occurrence of σ. The automaton T tracks and controls the behaviour of G. It changes state according to the events generated by G that the agent can observe and in turn, at each state x of T, the control rule $\psi(\sigma, x)$ dictates whether σ is to be enabled or disabled.

The behaviour of the closed-loop system, i.e., the sequences of events generated while the plant G is under the control of $\mathcal{S}$ (where $\mathcal{S} = (T, \psi)$), is represented by an automaton $\mathcal{S}/G$ whose closed behaviour, denoted by $L(\mathcal{S}/G)$, permits a string to be generated if the string is generated by G, generated by T and if each event in the string is enabled by ψ. The closed-loop system's marked behaviour is denoted by $L_m(\mathcal{S}/G)$ and consists of those strings in $L(\mathcal{S}/G)$ that are marked by both G and T.[1]

Typically, control problems will require finding supervisors that guarantee that the sequences generated or marked in the closed-loop system either equal some prescribed set of "legal" sequences or are a subset of these legal sequences. However, sometimes when a supervisor is attached to a plant there are sequences that can be generated by the plant and that, without control, would lead to

[1] In some of the literature, supervisors are not endowed with any ability to "mark" strings, i.e., they are either four-tuple automata with no specified set of marked states or effectively five-tuples whose set of marked states is equal to the entire set of states of the automaton. In those papers, the definition of closed-loop behaviour is slightly different than ours but all problems addressed using supervisors that have no ability to affect the marking of the closed-loop system can be recast as problems where supervisors do have a marking function, and vice versa.

marked states in the plant, but with control cannot reach a marked state. In other words, the supervision imposed prevents the system from reaching completion.[2] So, where possible, in the control problems below, we seek solutions that are *nonblocking*, i.e., every string generated by the closed-loop system can be completed to a marked string in the system. This requirement is expressed as follows: a supervisor $\mathcal{S}$ is *nonblocking for* G if

$$\overline{L_m(\mathcal{S}/G)} = L(\mathcal{S}/G),$$

where the overbar notation denotes prefix-closure. This is another departure from typical mathematical notation where the overbar might denote complement of a set or the negative of some logical proposition.

When a supervisor $\mathcal{S}$ ensures that $L_m(\mathcal{S}/G) = K$ (or $L(\mathcal{S}/G) = K$), we say that $\mathcal{S}$ *synthesizes* the language K, i.e., it guarantees that in closed-loop only the strings in K are recognized (respectively, generated).

When only one supervisor is used to control a discrete-event system, we are using *centralized* control. However, in some cases, we may require or find it favourable to use multiple, say n, supervisors to control a plant. We call this *decentralized control*. In such cases, the set of controllable events Σ_c can be subdivided into (not necessarily disjoint) subsets $\Sigma_{1,c}$, $\Sigma_{2,c}$, ..., $\Sigma_{n,c}$, where $\Sigma_{i,c}$ is the set of events that Supervisor i can disable. Similarly, the set Σ_o can be subdivided into (not necessarily disjoint) subsets $\Sigma_{1,o}$, $\Sigma_{2,o}$, ..., $\Sigma_{n,o}$, where $\Sigma_{i,o}$ is the set of events that Supervisor i can observe directly. We then extend the notion of a projection and supervisor so that P_i stands for the projection from $\Sigma^* \longrightarrow \Sigma_{i,o}^*$ and $\mathcal{S}_i = (T_i, \psi_i)$ stands for a supervisor that controls events in $\Sigma_{i,c}$ and observes events in $\Sigma_{i,o}$.

Since we will consider cases where multiple supervisors impose control on a plant, we require a way of formalizing the joint action of many controllers. The definition will capture the idea that an event in the plant will be disabled if any supervisor issues a disablement command. For simplicity, we restrict our discussions to the case of only two supervisors but the results in Section 3 generalize to an arbitrary, fixed number of supervisors. For supervisors $\mathcal{S}_1 = (T_1, \psi_1)$ and $\mathcal{S}_2 = (T_2, \psi_2)$ acting on G, the *conjunction* of $\mathcal{S}_1$ and $\mathcal{S}_2$ is the supervisor denoted by $\mathcal{S}_1 \wedge \mathcal{S}_2 = ((T_1 \times T_2), \psi_1 * \psi_2)$ where $T_1 \times T_2$ (called the *product*) recognizes the intersection of the languages recognized by T_1 and T_2 and $\psi_1 * \psi_2$ disables an event if and only if ψ_1 or ψ_2 disables it.

The behaviour of the closed-loop system when it is under the control of two supervisors, i.e., the sequences of events generated while the plant G is under the control of $\mathcal{S}_1 \wedge \mathcal{S}_2$ (where $\mathcal{S}_i = (T_i, \psi_i)$, $i = 1, 2$), is represented by an automaton $\mathcal{S}_1 \wedge \mathcal{S}_2/G$ whose closed behaviour, denoted by $L(\mathcal{S}_1 \wedge \mathcal{S}_2/G)$, permits a string to be generated if the string is generated by G, generated by T_i $(i = 1, 2)$ and if each event in the string is enabled by both ψ_1 and ψ_2. The closed-loop system's

[2] This scenario is not the same thing as the more commonly used property in computer science called *deadlock*. Deadlocks mean that a sequence reaches a state from which no other events may occur; the kind of "blocking" we have in mind arises when events can keep occurring but they will not lead the sequence to a marked state.

marked behaviour is denoted by $L_m(\mathcal{S}_1 \wedge \mathcal{S}_2/G)$ and consists of those strings in $L(\mathcal{S}_1 \wedge \mathcal{S}_2/G)$ that are marked by G and by each of T_1 and T_2.

3 Problems and Complexity

In this section we describe some of the key control problems in discrete-event systems theory. This list is highly noncomprehensive. We are focusing on the earlier work in the field, the work that set the stage for the more recent offshoots. Also, these problems are interesting because, when the languages in question are regular, in some cases the problems are solvable in polynomial time (in particular when all events can be observed by the supervisors); in some cases decidable but not solvable in polynomial time (presuming $P \neq NP$), which is the case when supervisors have only partial observation or are decentralized; and in a few surprising cases, undecidable (which is the case for small variants on the decidable decentralized DES problems).

Most of the DES control problems start with a plant G represented by an automaton and a formal language E that recognizes the legal sequences. This language E is called the *legal language*. The control problems involve finding a supervisor (or *supervisors* in the case of decentralized control) that manipulates the controllable events to guarantee that exactly the legal language is recognized by the closed-loop system. Sometimes such supervisors do not exist, i.e., there is no way to disable events at various points in the plant to guarantee that all the strings in E are generated—because allowing some string s in E to occur will necessitate allowing some other s' not in E to occur, if for example, $s' = s\sigma$ for some uncontrollable event σ and $s \in E$ but $s\sigma \notin E$. In those cases, we typically try to find supervisors that guarantee that a subset of the legal language is recognized in closed-loop and where possible, the largest subset of E that can safely be generated.

The following are the two main DES control problems considered in the 1980s:

Centralized Supervisory Control Problem 1. *Given a plant G over an alphabet Σ, a legal language E such that $\emptyset \neq E \subseteq L_m(G)$ and sets $\Sigma_c \subseteq \Sigma$, $\Sigma_o \subseteq \Sigma$, does there exist a nonblocking supervisor $\mathcal{S}$ for G such that*

$$L_m(\mathcal{S}/G) = E \ ?$$

The supervisor $\mathcal{S}$ can disable only events in Σ_c and can observe only the events in Σ_o. If a supervisor exists, construct it.

Centralized Supervisory Control Problem 2. *Given a plant G over an alphabet Σ, a prefix-closed legal language E such that $\emptyset \neq E \subseteq L(G)$, a minimally adequate language A such that $A \subseteq E$, and sets $\Sigma_c \subseteq \Sigma$, $\Sigma_o \subseteq \Sigma$, does there exist a nonblocking supervisor $\mathcal{S}$ for G such that*

$$A \subseteq L(\mathcal{S}/G) \subseteq E \ ?$$

The supervisor $\mathcal{S}$ can disable only events in Σ_c and can observe only the events in Σ_o. If a supervisor exists, construct it.

Problem 1 was first solved in [4]. Supervisors exist if and only if the system satisfies two properties: *controllability* and *observability*. Controllability was introduced in [5].

Definition 1 (Controllability). *Given G over Σ (with controllable events Σ_c). For a language $K \subseteq L(G)$, K is* controllable *with respect to G if*

$$\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K} \ .$$

If you think of E as some set of legal sequences and you want to know when it will be impossible to stop an illegal sequence from happening then E will need to be controllable. That is, if a string s starts out as legal (i.e., $s \in \overline{E}$), and an uncontrollable event σ *could* happen (i.e., $s\sigma \in L(G)$), then it must be the case that the uncontrollable event doesn't lead somewhere illegal (i.e., $s\sigma \in \overline{E}$)—since that event is uncontrollable and hence cannot be disabled. The definition of observability, adopted from [4], is as follows.

Definition 2 (Observability). *Given a plant G over alphabet Σ, sets $\Sigma_c, \Sigma_o \subseteq \Sigma$, projection $P : \Sigma^* \longrightarrow \Sigma_o^*$, $K \subseteq L_m(G)$ is* observable with respect to G, P *if for all $s, s' \in \Sigma^*$ such that $P(s) = P(s')$*

(i) $(\forall \sigma \in \Sigma)\ s'\sigma \in \overline{K} \wedge s \in \overline{K} \wedge s\sigma \in L(G) \Longrightarrow s\sigma \in \overline{K}$
(ii) $s' \in K \wedge s \in \overline{K} \cap L_m(G) \Longrightarrow s \in K$.

Intuitively, a supervisor knows what action to take if it knows what sequence of events actually occurred. However, a string which looks like (i.e., has the same projection as) another string may be potentially ambiguous in determining control action. On this basis, an informal description of observability is as follows. A language K is observable if (i) after the occurrence of an ambiguous string, s, in K, the decision to enable or disable a controllable event σ is forced by the action that a supervisor would take on other strings which look like s and (ii) the decision to mark or not mark a potentially confusing string is consistent for all strings that look alike to the supervisor.

If the entire legal language cannot be synthesized, typically one tries to synthesize a subset of E. If possible, one tries to find a supervisor whose closed-loop behaviour generates as many of the legal sequences as possible, so that the language $L(\mathcal{S}/G)$ is as large as possible and still contained in E. Such supervisors are called *maximally permissive* because they ensure that events are only disabled if necessary and, therefore, that the closed-loop language is as large as it can be without violating the legal language specification. Notice that in Centralized Problem 2, a language A is introduced (and was not present in the formulation of Centralized Problem 1). This is because if only some subset of the legal language is going to be generated in closed-loop, then the specification of A indicates some lower bound on acceptable behaviour.

When G is a finite automaton and E is a regular language, if all events are observable (i.e., $\Sigma_o = \Sigma$), Centralized Problem 2 is solvable in polynomial time as follows. Controllability is closed under arbitrary union and so, from lattice theory, the set of all controllable languages that are a subset of the legal language has a supremal element, denoted by $\sup \underline{C}(E, G)$. When G is a deterministic finite automaton (DFA) and E is a regular language, a DFA that recognizes $\sup \underline{C}(E, G)$ can be calculated in polynomial time [3]. First you check if $A \subseteq \sup \underline{C}(E, G)$; if not, no supervisor that solves Centralized Problem 2 exists. If the inclusion *is* satisfied, then you can construct a supervisor whose state transition structure is a recognizer for $\sup \underline{C}(E, G)$.

When some events are *not* observable, one can check whether the legal language is observable in polynomial time [6]. If E is both controllable and observable with respect to G, then one can construct a supervisor that solves Centralized Problem 1 by first constructing a DFA that recognizes E, then replacing all unobservable events in that automaton by ε, then doing an NFA-to-DFA conversion on the resulting NFA with ε moves. The resultant DFA recognizes $P(E)$, i.e., the set of legal sequences seen by a supervisor. One can then use this DFA to do supervision on the plant G to guarantee that only the strings of E are generated in closed-loop. The exponential-time operation cannot be obviated: as shown in [6], even if the system is both controllable and observable, one cannot always find a supervisor in polynomial time.

In contrast to the case when all events are observable (and one can find a supremal controllable sublanguage of E), observability is not closed under union and a supremal observable sublanguage of a given language need not exist. As a result, Centralized Problem 2 (with partial observation) was solved in [4] using the computation of the *infimal, prefix-closed, observable* superlanguage but this solution is not necessarily maximally permissive. It was shown in [7] that if G is a DFA and A is a regular language, then the infimal, prefix-closed, observable superlanguage of A, denoted by $\inf \underline{O}(A)$, can be calculated and is regular. It was shown in [4] that $\inf \underline{O}(A) \subseteq \sup \underline{C}(E)$ is a necessary and sufficient condition for there to exist a supervisor that satisfies Centralized Problem 2.

The following are the two main decentralized discrete-event control problems considered in the late 1980s and early 1990s:

Decentralized Supervisory Control Problem 1. *Given a plant G over an alphabet Σ, a legal language E such that $\emptyset \neq E \subseteq L_m(G)$ and sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, do there exist supervisors $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\mathcal{S}_1 \wedge \mathcal{S}_2$ is a nonblocking supervisor for G and such that*

$$L_m(\mathcal{S}_1 \wedge \mathcal{S}_2 / G) = E \ ?$$

For $i = 1, 2$, supervisor $\mathcal{S}_i$ can observe only events in $\Sigma_{i,o}$ and can control only events in $\Sigma_{i,c}$. The set of uncontrollable events is $\Sigma \setminus (\Sigma_{1,c} \cup \Sigma_{2,c})$. If supervisors exist, construct them.

Decentralized Supervisory Control Problem 2. *Given a plant G over an alphabet Σ, a prefix-closed legal language E such that $\emptyset \neq E \subseteq L(G)$, a minimally*

adequate language A such that $A \subseteq E$, and sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, do there exist supervisors $\mathcal{S}_1$ and $\mathcal{S}_2$ such that

$$A \subseteq L(\mathcal{S}_1 \wedge \mathcal{S}_2/G) \subseteq E \, ?$$

Here again, for $i = 1, 2$, supervisor $\mathcal{S}_i$ can observe only events in $\Sigma_{i,o}$ and can control only events in $\Sigma_{i,c}$. The set of uncontrollable events is $\Sigma \setminus (\Sigma_{1,c} \cup \Sigma_{2,c})$. If supervisors exist, construct them.

Decentralized Problem 1 was first solved in [8]. Supervisors exist if the system satisfies two properties: *controllability* and *co-observability*. The definition of co-observability, taken from [9], is as follows.

Definition 3 (Co-observability). *Given a plant G over alphabet Σ, sets $\Sigma_{1,c}$, $\Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, projections $P_1 : \Sigma^* \longrightarrow \Sigma_{1,o}^*$, $P_2 : \Sigma^* \longrightarrow \Sigma_{2,o}^*$, a language $K \subseteq L_m(G)$ is* co-observable with respect to G, P_1, P_2 *if for all $s, s', s'' \in \Sigma^*$, if $P_1(s) = P_1(s')$ and $P_2(s) = P_2(s'')$,*

$$\begin{array}{lll}
(\forall \sigma \in \Sigma_{1,c} \cap \Sigma_{2,c})\ s \in \overline{K} \wedge s\sigma \in L(G) \wedge s'\sigma, s''\sigma \in \overline{K} \Longrightarrow s\sigma \in \overline{K} & \textbf{conjunct 1} \\
\wedge\ (\forall \sigma \in \Sigma_{1,c} \setminus \Sigma_{2,c})\ s \in \overline{K} \wedge s\sigma \in L(G) \wedge s'\sigma \in \overline{K} \Longrightarrow s\sigma \in \overline{K} & \textbf{conjunct 2} \\
\wedge\ (\forall \sigma \in \Sigma_{2,c} \setminus \Sigma_{1,c})\ s \in \overline{K} \wedge s\sigma \in L(G) \wedge s''\sigma \in \overline{K} \Longrightarrow s\sigma \in \overline{K} & \textbf{conjunct 3} \\
\wedge\ s \in \overline{K} \cap L_m(G) \wedge s', s'' \in K \Longrightarrow s \in K. & \textbf{conjunct 4}
\end{array}$$

A string s that looks like s' to one supervisor and like s'' to another supervisor can cause potential control problems because if the sequence of events in s occurs, then both supervisors may not be sure whether s or a string that looks like it happened. With that in mind, an informal description of co-observability is as follows. A language K is co-observable if (1) after the occurrence of an ambiguous string, s, in K, the decision to enable or disable a controllable event σ is forced by the action that a supervisor which can control σ would take on other strings which look like s (encompassed by conjuncts 1–3 in the definition of co-observability), and (2) the decision to mark or not mark a potentially confusing string is determined by at least one of the supervisors (covered by conjunct 4).

It was shown in [10] that if G is finite-state and E is a regular language then co-observability can be checked in polynomial time. Decentralized Problem 2 was solved in [9] and requires the computation of the *infimal, prefix-closed, controllable and co-observable* language containing another language. It follows from the centralized control results of [6] that since the centralized analogue of Decentralized Problem 2 cannot be solved in polynomial time then *a fortiori* neither can the decentralized counterpart (otherwise one could use the decentralized solution where one of the two supervisors cannot control or observe any events and hence supervision falls entirely to one supervisor).

Decentralized DES formulations can be used to model telecommunication protocol problems. In [11], co-observability is used to verify that an erroneous variant of the Alternating Bit Protocol is incorrect. In [8, 12, 13], DES control and observation problems are used to model the data transmission problem for which the Alternating Bit Protocol is a solution.

3.1 Undecidable Problems

While the centralized and decentralized supervisory control problems presented in the last section were computable, variants of the decentralized control problem are not decidable. Interest in such DES problems have come both from within the DES control systems community and from the computer science community. In particular, two research groups independently produced results that show that variants on Problem 2 are undecidable. In [14], it is shown that the following problem is undecidable.

Decentralized Supervisory Control Problem 3. *Given a finite-state plant G over an alphabet Σ and sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, do there exist supervisors $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\mathcal{S}_1 \wedge \mathcal{S}_2$ is a nonblocking supervisor for G?*

The proof in [14] shows that the Halting Problem on Turing machines can be reduced to Decentralized Problem 3. On the face of it, Decentralized Problem 3 appears to be a special case of Decentralized Problem 4 (below), which is also shown to be undecidable in [15, 16]. In fact, one can show that each problem reduces to the other (because the additional legal language requirement of Decentralized Problem 4 can be incorporated into the plant itself).

Decentralized Supervisory Control Problem 4. *Given a finite-state plant G over an alphabet Σ, a regular language E, and sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, do there exist supervisors $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\mathcal{S}_1 \wedge \mathcal{S}_2$ is a nonblocking supervisor for G and such that*

$$L_m(\mathcal{S}_1 \wedge \mathcal{S}_2/G) \subseteq E \ ?$$

The proof in [16] that Decentralized Problem 4 is undecidable reduces a decentralized observation problem to the decentralized control problem. The latter problem is shown to be undecidable by reducing the Post Correspondence Problem (PCP), which is known to be undecidable, to the observation problem.

Since there are limitations on what can be achieved by decentralized controllers that make independent control decisions to effect joint control, various research groups have also explored decentralized DES problems where the supervisors may communicate with each other. In [17], it is shown that for a decentralized control problem where supervisors may communicate but where communication delays are unbounded, checking for existence of supervisors that solve the problem is undecidable. On the other hand, in the paper it is also shown that a decentralized observation problem with bounded-delay communication is decidable.

4 Discussion

Early applications of discrete-event systems control theory had an industrial engineering flavour to them, with a focus on manufacturing systems; see [2, 18–20] for a sampling. In those early years of DES theory, most of the control was centralized and events were generally assumed to be fully observable. Although,

as discussed in Section 3, centralized DES problems (involving finite-state plants and regular-language specifications) can be solved in polynomial time, often these monolithic plants are the result of a composition of multiple subplants. The typical composition strategies used create larger DFAs whose state spaces are the Cartesian product of the state spaces of their constituent subplants. Consequently, the number of their states may be exponential in the number of subplants. As a result, the second wave of DES work focused on modular architectures such as decentralized control [8, 9, 21] and hierarchical control [22, 23] as a strategy for managing the computational complexity of centralized systems. However, as pointed out in Section 3, when multiple supervisors are involved, each agent no longer retains a full view of all event strings and the resulting partial observability leads to computational pitfalls which are at best exponential in the state space of each constituent module (which would at least be smaller than a single, centralized module) but at worst result in undecidable problems.

The state-space explosion problem coupled with the restrictions of the basic DES model has limited the applicability of the research to real engineering problems. Researchers have sought to address some of these limitations and to generally enhance the framework by augmenting the basic DES model to accommodate timed events [24], by augmenting the model to allow for events to have probabilities associated with them [25, 26], and by using formalisms such as Petri nets (which can characterize classes of languages that are not regular) [27], temporal logic [28] or other modal logics [29, 30]. Another promising avenue of research is the *limited lookahead* approach [31] whereby instead of storing the entire automaton representing the plant and the entire automaton the accepts the legal language, we keep a running window of strings of some fixed number N and then the supervisor performs on-line control and after each event occurrence, the window is updated.

Our current research seeks to address some of the limitations of the basic DES model by combining a Petri net representation of plants together with a limited lookahead approach to traversing the reachability graph of the Petri net. We have been applying this approach to automate concurrency control in software engineering [32]. The idea of combining DES control and software engineering has also been the focus of another research group that includes industrial partners [33]. In addition, we have been trying to model emergency response protocols as discrete-event systems. To accomplish this, we have had to augment the standard Petri net model for DESs to account for event transitions with various timing properties and event transition choices that occur with some probability [34].

There has been very little interaction between the automata theory and DES communities even though the latter could benefit from the theoretical developments and insights of the former. Perhaps too, the control-theoretic angle and recent interest in the DES community in solving real-world problems could provide some motivating examples for the automata theory community.

Acknowledgements. I would like to thank Mike Domaratzki, Kai Salomaa and John Thistle who provided helpful feedback on an earlier draft of this paper.

References

1. Ramadge, P.J.: Control and Supervision of Discrete Event Processes. PhD thesis, Department of Electrical Engineering, University of Toronto (1983)
2. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete-event processes. SIAM Journal of Control and Optimization 25(1), 206–230 (1987)
3. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, 2nd edn. Springer, New York (2008)
4. Lin, F., Wonham, W.M.: On observability of discrete-event systems. Information Sciences 44, 173–198 (1988)
5. Ramadge, P.J., Wonham, W.M.: Supervision of discrete event processes. In: Proceedings of the 21st IEEE Conference on Decision and Control, vol. 3, pp. 1228–1229 (December 1982)
6. Tsitsiklis, J.N.: On the control of discrete-event dynamical systems. Mathematics of Control, Signals, and Systems 2, 95–107 (1989)
7. Rudie, K., Wonham, W.M.: The infimal prefix-closed and observable superlanguage of a given language. Systems & Control Letters 15(5), 361–371 (1990)
8. Cieslak, R., Desclaux, C., Fawaz, A.S., Varaiya, P.: Supervisory control of discrete-event processes with partial observations. IEEE Transactions on Automatic Control 33(3), 249–260 (1988)
9. Rudie, K., Wonham, W.M.: Think globally, act locally: Decentralized supervisory control. IEEE Transactions on Automatic Control 37(11), 1692–1708 (1992)
10. Rudie, K., Willems, J.C.: The computational complexity of decentralized discrete-event control problems. IEEE Transactions on Automatic Control 40(7), 1313–1319 (1995)
11. Rudie, K., Wonham, W.M.: Protocol verification using discrete-event systems. In: Proceedings of the 31st IEEE Conference on Decision and Control, Tucson, Arizona, pp. 3770–3777 (December 1992)
12. Rudie, K., Wonham, W.M.: Supervisory control of communicating processes. In: Logrippo, L., Probert, R.L., Ural, H. (eds.) Protocol Specification, Testing and Verification X, pp. 243–257. Elsevier Science (North-Holland), Amsterdam (1990); Expanded version appears as Systems Control Group Report #8907, Department of Electrical Engineering, University of Toronto (1989)
13. Puri, A., Tripakis, S., Varaiya, P.: Problems and examples of decentralized observation and control for discrete event systems. In: Caillaud, B., Darondeau, P., Lavagno, L., Xie, X. (eds.) Synthesis and Control of Discrete Event Systems, pp. 37–55. Kluwer Academic Publishers, Dordrecht (2001)
14. Thistle, J.G.: Undecidability in decentralized supervision. Systems & Control Letters 54, 503–509 (2005)
15. Tripakis, S.: Undecidable problems of decentralized observation and control. In: Proceedings of the IEEE Conference on Decision and Control, Orlando, FL, pp. 4104–4109 (December 2001)
16. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Information Processing Letters 90, 21–28 (2004)
17. Tripakis, S.: Decentralized control of discrete-event systems with bounded or unbounded delay communication. IEEE Transactions on Automatic Control 49(9), 1489–1501 (2004)
18. Brandin, B.A., Wonham, W.M., Benhabib, B.: Manufacturing cell supervisory control—a timed discrete-event system approach. In: Proceedings of the IEEE Conference on Robotics and Automation, Nice, France, pp. 931–936 (May 1992)

19. Krogh, B.H., Holloway, L.E.: Synthesis of feedback control logic for discrete manufacturing systems. Automatica 27(4), 641–651 (1991)
20. Balemi, S., Hoffmann, G.J., Gyugyi, P., Wong-Toi, H., Franklin, G.F.: Supervisory control of a rapid thermal multiprocessor. IEEE Transactions on Automatic Control 38(7), 1040–1059 (1993)
21. Yoo, T.-S., Lafortune, S.: A general architecture for decentralized supervisory control of discrete-event systems. Discrete Event Dynamic Systems: Theory and Applications 12, 335–377 (2002)
22. Zhong, H., Wonham, W.M.: On the consistency of hierarchical supervision in discrete-event systems. IEEE Transactions on Automatic Control 35(10), 1125–1134 (1990)
23. Caines, P.E., Wei, Y.J.: The hierarchical lattices of a finite machine. Systems & Control Letters 25, 257–263 (1995)
24. Brandin, B.A., Wonham, W.M.: Supervisory control of timed discrete-event systems. IEEE Transactions on Automatic Control 39(2), 329–342 (1994)
25. Lawford, M., Wonham, W.M.: Supervisory control of probabilistic discrete event systems. In: Proceedings of the 36th Midwest Symposium on Circuits and Systems, Detroit, MI, pp. 327–331 (1993)
26. Kumar, R., Garg, V.K.: Control of stochastic discrete event systems modeled by probabilistic languages. IEEE Transactions on Automatic Control 46(4), 593–606 (2001)
27. Holloway, L., Krogh, B.H., Giua, A.: A survey of Petri net methods for controlled discrete event systems. Discrete Event Dynamic Systems: Theory and Applications 2(7), 151–190 (1997)
28. Ostroff, J.S., Wonham, W.M.: A framework for real-time discrete event control. IEEE Transactions on Automatic Control 35(4), 386–397 (1990)
29. Ricker, S.L., Rudie, K.: Know means no: Incorporating knowledge into decentralized discrete-event control. IEEE Transactions on Automatic Control 45(9), 1656–1668 (2000)
30. Ricker, S.L., Rudie, K.: Knowledge is a terrible thing to waste: Using inference in discrete-event contol problems. IEEE Transactions on Automatic Control 52(3), 428–441 (2007)
31. Chung, S.-L., Lafortune, S., Lin, F.: Limited lookahead policies in supervisory control of discrete event systems. IEEE Transactions on Automatic Control 37(12), 1921–1935 (1992)
32. Auer, A., Dingel, J., Rudie, K.: Concurrency control generation for dynamic threads using discrete-event systems. In: Proceedings of the Allerton Conference on Communication, Control and Computing, Monticello, IL, September 30-October 2, pp. 927–934 (2009)
33. Kelly, T., Wang, Y., Lafortune, S., Mahlke, S.: Eliminating concurrency bugs with control engineering. Computer 42(12), 52–60 (2009)
34. Whittaker, S.-J., Rudie, K., McLellan, J., Haar, S.: Choice-point nets: A discrete-event modelling technique for analyzing health care protocols. In: Proceedings of the Allerton Conference on Communication, Control and Computing, Monticello, IL, September 30-October 2, pp. 652–659 (2009)

Large-Scale Training of SVMs with Automata Kernels

Cyril Allauzen[1], Corinna Cortes[1], and Mehryar Mohri[2,1]

[1] Google Research, 76 Ninth Avenue, New York, NY 10011, USA
[2] Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, NY 10012, USA

Abstract. This paper presents a novel application of automata algorithms to machine learning. It introduces the first optimization solution for support vector machines used with sequence kernels that is purely based on weighted automata and transducer algorithms, without requiring any specific solver. The algorithms presented apply to a family of kernels covering all those commonly used in text and speech processing or computational biology. We show that these algorithms have significantly better computational complexity than previous ones and report the results of large-scale experiments demonstrating a dramatic reduction of the training time, typically by several orders of magnitude.

1 Introduction

Weighted automata and transducer algorithms have been used successfully in a variety of natural language processing applications, including speech recognition, speech synthesis, and machine translation [17]. More recently, they have found other important applications in machine learning [5,1]: they can be used to define a family of sequence kernels, *rational kernels* [5], which covers all sequence kernels commonly used in machine learning applications in bioinformatics or text and speech processing.

Sequences kernels are similarity measures between sequences that are positive definite symmetric, which implies that their value coincides with an inner product in some Hilbert space. Kernels are combined with effective learning algorithms such as support vector machines (SVMs) [6] to create powerful classification techniques, or with other learning algorithms to design regression, ranking, clustering, or dimensionality reduction solutions [19]. These kernel methods are among the most widely used techniques in machine learning.

Scaling these algorithms to large-scale problems remains computationally challenging however, both in time and space. One solution consists of using approximation techniques for the kernel matrix, e.g., [9,2,21,13] or to use early stopping for optimization algorithms [20]. However, these approximations can of course result in some loss in accuracy, which, depending on the size of the training data and the difficulty of the task, can be significant.

This paper presents general techniques for speeding up large-scale SVM training when used with an arbitrary rational kernel, without resorting to such approximations. We show that coordinate descent approaches similar to those used by [10] for linear kernels can be extended to SVMs combined with rational kernels to design faster algorithms with significantly better computational complexity. Remarkably, our solution techniques are purely based on weighted automata and transducer algorithms and require no specific optimization solver. To the best of our knowledge, they form the first

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 17–27, 2011.

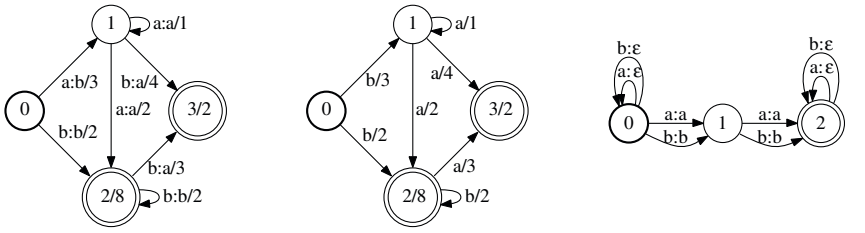


Fig. 1. (a) Example of weighted transducer $\mathbf{U}$. (b) Example of weighted automaton $\mathbf{A}$. In this example, $\mathbf{A}$ can be obtained from $\mathbf{U}$ by projection on the output and $\mathbf{U}(aab, baa) = \mathbf{A}(baa) = 3\times1\times4\times2+3\times2\times3\times2$. (c) Bigram counting transducer $\mathbf{T}_2$ for $\Sigma = \{a, b\}$. Initial states are represented by bold circles, final states by double circles and the weights of transitions and final states are indicated after the slash separator.

automata-based optimization algorithm of SVMs, probably the most widely used algorithm in machine learning. Furthermore, we show experimentally that our techniques lead to a dramatic speed-up of training with sequence kernels. In most cases, we observe an improvement by several orders of magnitude.

The remainder of the paper is structured as follows. We start with a brief introduction to weighted transducers and rational kernels (Section 2), including definitions and properties relevant to the following sections. Section 3 provides a short introduction to kernel methods such as SVMs and presents an overview of the coordinate descent solution by [10] for linear SVMs. Section 4 shows how a similar solution can be derived in the case of rational kernels. The analysis of the complexity and the implementation of this technique are described and discussed in Section 5. In section 6, we report the results of experiments with a large dataset and with several types of kernels demonstrating the substantial reduction of training time using our techniques.

2 Preliminaries

This section introduces the essential concepts and definitions related to weighted transducers and rational kernels. Generally, we adopt the definitions and terminology of [5].

Weighted transducers are finite-state transducers in which each transition carries some weight in addition to the input and output labels. The weight set has the structure of a semiring [12]. In this paper, we only consider weighted transducers over the *real semiring* $(\mathbb{R}_+, +, \times, 0, 1)$. Figure 1(a) shows an example. A path from an initial state to a final state is an accepting path. The input (resp. output) label of an accepting path is obtained by concatenating together the input (resp. output) symbols along the path from the initial to the final state. Its weight is computed by multiplying the weights of its constituent transitions and multiplying this product by the weight of the initial state of the path (which equals one in our work) and by the weight of the final state of the path. The weight associated by a weighted transducer $\mathbf{U}$ to a pair of strings $(\mathbf{x}, \mathbf{y}) \in \Sigma^* \times \Sigma^*$ is denoted by $\mathbf{U}(\mathbf{x}, \mathbf{y})$. For any transducer $\mathbf{U}$ we define the linear operator D as the sum of the weights of all accepting paths of $\mathbf{U}$.

A *weighted automaton* $\mathbf{A}$ can be defined as a weighted transducer with identical input and output labels. Discarding the input labels of a weighted transducer $\mathbf{U}$ results in a weighted automaton $\mathbf{A}$, said to be the *output projection of* $\mathbf{U}$, $\mathbf{A} = \Pi_2(\mathbf{U})$. The automaton in Figure 1(b) is the output projection of the transducer in Figure 1(a).

The standard operations of sum $+$, product or concatenation $\cdot$, multiplication by a real number and Kleene-closure * are defined for weighted transducers [18]. The *inverse* of a transducer $\mathbf{U}$, denoted by $\mathbf{U}^{-1}$, is obtained by swapping the input and output labels of each transition. For all pairs of strings $(\mathbf{x}, \mathbf{y})$, we have $\mathbf{U}^{-1}(\mathbf{x}, \mathbf{y}) = \mathbf{U}(\mathbf{y}, \mathbf{x})$. The *composition* of two weighted transducers $\mathbf{U}_1$ and $\mathbf{U}_2$ with matching output and input alphabets Σ, is a weighted transducer denoted by $\mathbf{U}_1 \circ \mathbf{U}_2$ when the sum:

$$(\mathbf{U}_1 \circ \mathbf{U}_2)(\mathbf{x}, \mathbf{y}) = \sum_{\mathbf{z} \in \Sigma^*} \mathbf{U}_1(\mathbf{x}, \mathbf{z}) \times \mathbf{U}_2(\mathbf{z}, \mathbf{y})$$

is well-defined and in $\mathbb{R}$ for all $\mathbf{x}, \mathbf{y}$ [18]. It can be computed in time $O(|\mathbf{U}_1||\mathbf{U}_2|))$ where $|\mathbf{U}|$ denotes the sum of the number of states and transitions of a transducer $\mathbf{U}$.

Given a non-empty set X, a function $K: X \times X \to \mathbb{R}$ is called a *kernel*. K is said to be *positive definite symmetric* (PDS) when the matrix $(K(\mathbf{x}_i, \mathbf{x}_j))_{1 \leq i,j \leq m}$ is symmetric and positive semi-definite (PSD) for any choice of m points in X. A kernel between sequences $K: \Sigma^* \times \Sigma^* \to \mathbb{R}$ is *rational* [5] if there exists a weighted transducer $\mathbf{U}$ such that K coincides with the function defined by $\mathbf{U}$, that is $K(\mathbf{x}, \mathbf{y}) = \mathbf{U}(\mathbf{x}, \mathbf{y})$ for all $\mathbf{x}, \mathbf{y} \in \Sigma^*$. When there exists a weighted transducer $\mathbf{T}$ such that $\mathbf{U}$ can be decomposed as $\mathbf{U} = \mathbf{T} \circ \mathbf{T}^{-1}$, then it was shown by [5] that K is PDS. All the sequence kernels seen in practice are precisely PDS rational kernels of this form.

A standard family of rational kernels is n-gram kernels, see e.g. [15, 14]. Let $c_{\mathbf{x}}(\mathbf{z})$ be the number of occurrences of $\mathbf{z}$ in $\mathbf{x}$. The n-gram kernel K_n of order n is defined as $K_n(\mathbf{x}, \mathbf{y}) = \sum_{|\mathbf{z}|=n} c_{\mathbf{x}}(\mathbf{z}) c_{\mathbf{y}}(\mathbf{z})$. K_n is a PDS rational kernel since it corresponds to the weighted transducer $\mathbf{T}_n \circ \mathbf{T}_n^{-1}$ where the transducer $\mathbf{T}_n$ is defined such that $\mathbf{T}_n(\mathbf{x}, \mathbf{z}) = c_x(\mathbf{z})$ for all $\mathbf{x}, \mathbf{z} \in \Sigma^*$ with $|\mathbf{z}| = n$. The transducer $\mathbf{T}_2$ for $\Sigma = \{a, b\}$ is shown in Figure 1(c).

3 Kernel Methods and SVM Optimization

Kernel methods are widely used in machine learning. They have been successfully used in a variety of learning tasks including classification, regression, ranking, clustering, and dimensionality reduction. This section gives a brief overview of these methods, and discusses in more detail one of the most popular kernel learning algorithms, SVMs.

3.1 Overview of Kernel Methods

Complex learning tasks are often tackled using a large number of features. Each point of the input space X is mapped to a high-dimensional feature space F via a non-linear mapping $\mathbf{\Phi}$. This may be to seek a linear separation in a higher-dimensional space, which was not achievable in the original space, or to exploit other regression, ranking, clustering, or manifold properties that are easier to attain in that space. The dimension of the feature space F can be very large. In document classification, the features may be the set of all trigrams. Thus, even for a vocabulary of just 200,000 words, the dimension of F is 2×10^{15}.

The high dimensionality of F does not necessarily affect the generalization ability of large-margin algorithms such as SVMs: remarkably, these algorithms benefit from theoretical guarantees for good generalization that depend only on the number of training points and the separation *margin*, and not on the dimensionality of the feature space. But the high dimensionality of F can directly impact the efficiency and even the practicality of such learning algorithms, as well as their use in prediction. This is because to determine their output hypothesis or for prediction, these learning algorithms rely on the computation of a large number of dot products in the feature space F.

A solution to this problem is the so-called *kernel method*. This consists of defining a function $K: X \times X \to \mathbb{R}$ called a *kernel*, such that the value it associates to two examples $\mathbf{x}$ and $\mathbf{y}$ in input space, $K(\mathbf{x}, \mathbf{y})$, coincides with the dot product of their images $\mathbf{\Phi}(\mathbf{x})$ and $\mathbf{\Phi}(\mathbf{y})$ in feature space. K is often viewed as a similarity measure:

$$\forall \mathbf{x}, \mathbf{y} \in X, \quad K(\mathbf{x}, \mathbf{y}) = \mathbf{\Phi}(\mathbf{x})^\top \mathbf{\Phi}(\mathbf{y}). \tag{1}$$

A crucial advantage of K is efficiency: there is no need anymore to define and explicitly compute $\mathbf{\Phi}(\mathbf{x})$, $\mathbf{\Phi}(\mathbf{y})$, and $\mathbf{\Phi}(\mathbf{x})^\top \mathbf{\Phi}(\mathbf{y})$. Another benefit of K is flexibility: K can be arbitrarily chosen so long as the existence of $\mathbf{\Phi}$ is guaranteed, a condition that holds when K verifies Mercer's condition. This condition is important to guarantee the convergence of training for algorithms such as SVMs. In the discrete case, it is equivalent to K being PDS.

One of the most widely used two-group classification algorithm is SVMs [6]. The version of SVMs without offsets is defined via the following convex optimization problem for a training sample of m points $\mathbf{x}_i \in X$ with labels $y_i \in \{1, -1\}$:

$$\min_{\mathbf{w}, \boldsymbol{\xi}} \frac{1}{2}\mathbf{w}^2 + C \sum_{i=1}^{m} \xi_i \quad \text{s.t.} \quad y_i \mathbf{w}^\top \mathbf{\Phi}(\mathbf{x}_i) \geq 1 - \xi_i \quad \forall i \in [1, m],$$

where the vector $\mathbf{w}$ defines a hyperplane in the feature space, $\boldsymbol{\xi}$ is the m-dimensional vector of slack variables, and $C \in \mathbb{R}_+$ is a trade-off parameter. The problem is typically solved by introducing Lagrange multipliers $\boldsymbol{\alpha} \in \mathbb{R}^m$ for the set of constraints. The standard dual optimization for SVMs can be written as the convex optimization problem:

$$\min_{\boldsymbol{\alpha}} \quad F(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^\top \mathbf{Q} \boldsymbol{\alpha} - \mathbf{1}^\top \boldsymbol{\alpha} \quad \text{s.t.} \quad \mathbf{0} \leq \boldsymbol{\alpha} \leq \mathbf{C},$$

where $\boldsymbol{\alpha} \in \mathbb{R}^m$ is the vector of dual variables and the PSD matrix $\mathbf{Q}$ is defined in terms of the kernel matrix $\mathbf{K}$: $\mathbf{Q}_{ij} = y_i y_j \mathbf{K}_{ij} = y_i y_j \mathbf{\Phi}(\mathbf{x}_i)^\top \mathbf{\Phi}(\mathbf{x}_j)$, $i, j \in [1, m]$. Expressed with the dual variables, the solution vector $\mathbf{w}$ can be written as $\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{\Phi}(\mathbf{x}_i)$.

3.2 Coordinate Descent Solution for SVM Optimization

A straightforward way to solve the convex dual SVM problem is to use a coordinate descent method and to update only one coordinate α_i at each iteration, see [10]. The optimal step size $\beta^\star$ corresponding to the update of α_i is obtained by solving

$$\min_{\beta} \quad \frac{1}{2}(\boldsymbol{\alpha} + \beta \mathbf{e}_i)^\top \mathbf{Q}(\boldsymbol{\alpha} + \beta \mathbf{e}_i) - \mathbf{1}^\top(\boldsymbol{\alpha} + \beta \mathbf{e}_i) \quad \text{s.t.} \quad \mathbf{0} \leq \boldsymbol{\alpha} + \beta \mathbf{e}_i \leq \mathbf{C},$$

SVMCOORDINATEDESCENT($(\mathbf{x}_i)_{i\in[1,m]}$)
1 $\boldsymbol{\alpha} \leftarrow \mathbf{0}$
2 **while** $\boldsymbol{\alpha}$ not optimal **do**
3 **for** $i \in [1, m]$ **do**
4 $g \leftarrow y_i \mathbf{x}_i^\top \mathbf{w} - 1$ and $\alpha_i' \leftarrow \min(\max(\alpha_i - \frac{g}{\mathbf{Q}_{ii}}, 0), C)$
5 $\mathbf{w} \leftarrow \mathbf{w} + (\alpha_i' - \alpha_i)\mathbf{x}_i$ and $\alpha_i \leftarrow \alpha_i'$
6 **return** $\mathbf{w}$

Fig. 2. Coordinate descent solution for SVM

where $\boldsymbol{e}_i$ is an m-dimensional unit vector. Ignoring constant terms, the optimization problem can be written as

$$\min_{\beta} \quad \frac{1}{2}\beta^2 \mathbf{Q}_{ii} + \beta \boldsymbol{e}_i^\top (\mathbf{Q}\boldsymbol{\alpha} - \mathbf{1}) \quad \text{s.t.} \quad 0 \leq \alpha_i + \beta \leq C.$$

If $\mathbf{Q}_{ii} = \boldsymbol{\Phi}(\mathbf{x}_i)^\top \boldsymbol{\Phi}(\mathbf{x}_i) = 0$, then $\boldsymbol{\Phi}(\mathbf{x}_i) = \mathbf{0}$ and $\mathbf{Q}_i = \boldsymbol{e}_i^\top \mathbf{Q} = \mathbf{0}$. Hence the objective function reduces to $-\beta$, and the optimal step size is $\beta^\star = C - \alpha_i$, resulting in the update: $\alpha_i \leftarrow 0$. Otherwise $\mathbf{Q}_{ii} \neq 0$ and the objective function is a second-degree polynomial in β. Let $\beta_0 = -\frac{\mathbf{Q}_i^\top \boldsymbol{\alpha} - 1}{\mathbf{Q}_{ii}}$, then the optimal step size and update is given by

$$\beta^\star = \begin{cases} \beta_0, & \text{if } -\alpha_i \leq \beta_0 \leq C - \alpha_i, \\ -\alpha_i, & \text{if } \beta_0 \leq -\alpha_i, \\ C - \alpha_i, & \text{otherwise} \end{cases} \quad \text{and } \alpha_i \leftarrow \min\left(\max\left(\alpha_i - \frac{\mathbf{Q}_i^\top \boldsymbol{\alpha} - 1}{\mathbf{Q}_{ii}}, 0\right), C\right).$$

When the matrix $\mathbf{Q}$ is too large to store in memory and $\mathbf{Q}_{ii} \neq 0$, the vector $\mathbf{Q}_i$ must be computed at each update of α_i. If the cost of the computation of each entry $\mathbf{K}_{ij}$ is in $O(N)$ where N is the dimension of the feature space, computing $\mathbf{Q}_i$ is in the $O(mN)$, and hence the cost of each update is in $O(mN)$.

The choice of the coordinate α_i to update is based on the gradient. The gradient of the objective function is $\nabla F(\boldsymbol{\alpha}) = \mathbf{Q}\boldsymbol{\alpha} - \mathbf{1}$. At a cost in $O(mN)$ it can be updated via

$$\nabla F(\boldsymbol{\alpha}) \leftarrow \nabla F(\boldsymbol{\alpha}) + \Delta(\alpha_i)\mathbf{Q}_i.$$

Hsieh et al. [10] observed that when the kernel is linear, $\mathbf{Q}_i^\top \boldsymbol{\alpha}$ can be expressed in terms of $\mathbf{w}$, the SVM weight vector solution, $\mathbf{w} = \sum_{j=1}^m y_j \alpha_j \mathbf{x}_j$:

$$\mathbf{Q}_i^\top \boldsymbol{\alpha} = \sum_{j=1}^m y_i y_j (\mathbf{x}_i^\top \mathbf{x}_j)\alpha_j = y_i \mathbf{x}_i^\top \mathbf{w}.$$

If the weight vector $\mathbf{w}$ is maintained throughout the iterations, then the cost of an update is only in $O(N)$ in this case. The weight vector $\mathbf{w}$ can be updated via

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta(\alpha_i) y_i \mathbf{x}_i.$$

Maintaining the gradient $\nabla F(\boldsymbol{\alpha})$ is however still costly. The jth component of the gradient can be expressed as follows:

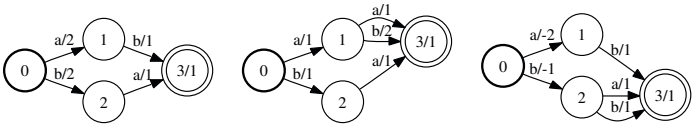
$$[\nabla F(\boldsymbol{\alpha})]_j = [Q\boldsymbol{\alpha} - \mathbf{1}]_j = \sum_{i=1}^m y_i y_j \mathbf{x}_i^\top \mathbf{x}_j \alpha_i - 1 = \mathbf{w}^\top (y_j \mathbf{x}_j) - 1.$$

SVMRATIONALKERNELS($(\mathbf{\Phi}'_i)_{i\in[1,m]}$)
1 $\boldsymbol{\alpha} \leftarrow \mathbf{0}$
2 **while** $\boldsymbol{\alpha}$ not optimal **do**
3 **for** $i \in [1, m]$ **do**
4 $g \leftarrow \mathrm{D}(\mathbf{\Phi}'_i \circ \mathbf{W}') - 1$ and $\alpha'_i \leftarrow \min(\max(\alpha_i - \frac{g}{\mathbf{Q}_{ii}}, 0), C)$
5 $\mathbf{W}' \leftarrow \mathbf{W}' + (\alpha'_i - \alpha_i)\mathbf{\Phi}'_i$ and $\alpha_i \leftarrow \alpha'_i$
6 **return** $\mathbf{W}'$

Fig. 3. Coordinate descent solution for rational kernels

i	$\mathbf{x}_i$	y_i	$\mathbf{Q}_{ii}$
1	$ababa$	$+1$	8
2	$abaab$	$+1$	6
3	$abbab$	-1	6

(a)



(b) (c) (d)

Fig. 4. (a) Example dataset. (b-d) The automata $\mathbf{\Phi}'_i$ corresponding to the dataset of (a) when using a bigram kernel. The given $\mathbf{\Phi}'_i$ and $\mathbf{Q}_{ii}$'s assume the use of a bigram kernel.

The update for the main term of component j of the gradient is thus given by:

$$\mathbf{w}^\top \mathbf{x}_j \leftarrow \mathbf{w}^\top \mathbf{x}_j + (\Delta\mathbf{w})^\top \mathbf{x}_j.$$

Each of these updates can be done in $O(N)$. The full update for the gradient can hence be done in $O(mN)$. Several heuristics can be used to eliminate the cost of maintaining the gradient. For instance, one can choose a random α_i to update at each iteration [10] or sequentially update the α_is. Hsieh et al. [10] also showed that it is possible to use the chunking method of [11] in conjunction with such heuristics. Using the results from [16], [10] showed that the resulting coordinate descent algorithm, SVMCOORDINATEDESCENT (Figure 2) converges to the optimal solution with a linear or faster convergence rate.

4 Coordinate Descent Solution for Rational Kernels

This section shows that, remarkably, coordinate descent techniques similar to those described in the previous section can be used in the case of rational kernels.

For rational kernels, the input "vectors" $\mathbf{x}_i$ are sequences, or distributions over sequences, and the expression $\sum_{j=1}^m y_j \alpha_j \mathbf{x}_j$ can be interpreted as a weighted regular expression. For any $i \in [1, m]$, let $\mathbf{X}_i$ be a simple weighted automaton representing $\mathbf{x}_i$, and let $\mathbf{W}$ denote a weighted automaton representing $\mathbf{w} = \sum_{j=1}^m y_j \alpha_j \mathbf{x}_j$. Let $\mathbf{U}$ be the weighted transducer associated to the rational kernel K. Using the linearity of D and distributivity properties just presented, we can now write:

$$\mathbf{Q}_i^\top \boldsymbol{\alpha} = \sum_{j=1}^m y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)\alpha_j = \sum_{j=1}^m y_i y_j \,\mathrm{D}(\mathbf{X}_i \circ \mathbf{U} \circ \mathbf{X}_j)\alpha_j \tag{2}$$

$$= \mathrm{D}(y_i \mathbf{X}_i \circ \mathbf{U} \circ \sum_{j=1}^m y_j \alpha_j \mathbf{X}_j) = \mathrm{D}(y_i \mathbf{X}_i \circ \mathbf{U} \circ \mathbf{W}).$$

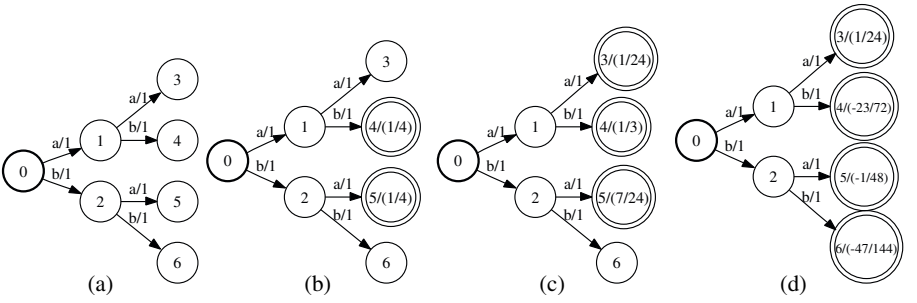


Fig. 5. Evolution of $\mathbf{W}'$ through the first iteration of SVMRATIONALKERNELS on the dataset from Figure 4

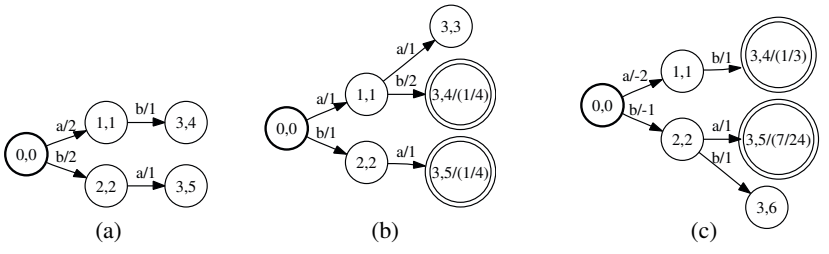


Fig. 6. The automata $\mathbf{\Phi}'_i \circ \mathbf{W}'$ during the first iteration of SVMRATIONALKERNELS on the data in Figure 4

Since $\mathbf{U}$ is a constant, in view of the complexity of composition, the expression $y_i \mathbf{X}_i \circ \mathbf{U} \circ \mathbf{W}$ can be computed in time $O(|\mathbf{X}_i||\mathbf{W}|)$. When $y_i \mathbf{X}_i \circ \mathbf{U} \circ \mathbf{W}$ is acyclic, which is the case for example if $\mathbf{U}$ admits no input ϵ-cycle, then $\mathrm{D}(y_i \mathbf{X}_i \circ \mathbf{U} \circ \mathbf{W})$ can be computed in linear time in the size of $y_i \mathbf{X}_i \circ \mathbf{U} \circ \mathbf{W}$ using a shortest-distance algorithm, or forward-backward algorithm. For all of the rational kernels that we are aware of, $\mathbf{U}$ admits no input ϵ-cycle and this property holds. Thus, in that case, if we maintain a weighted automaton $\mathbf{W}$ representing $\mathbf{w}$, $\mathbf{Q}_i^\top \boldsymbol{\alpha}$ can be computed in $O(|\mathbf{X}_i||\mathbf{W}|)$. This complexity does not depend on m and the explicit computation of m kernel values $K(\mathbf{x}_i, \mathbf{x}_j)$, $j \in [1, m]$, is avoided. The update rule for $\mathbf{W}$ consists of augmenting the weight of sequence $\mathbf{x}_i$ in the weighted automaton by $\Delta(\alpha_i) y_i$:

$$\mathbf{W} \leftarrow \mathbf{W} + \Delta(\alpha_i) y_i \mathbf{X}_i.$$

This update can be done very efficiently if $\mathbf{W}$ is deterministic, in particular if it is represented as a deterministic trie.

When the weighted transducer $\mathbf{U}$ can be decomposed as $\mathbf{T} \circ \mathbf{T}^{-1}$, as for all sequence kernels seen in practice, we can further improve the form of the updates. Let $\Pi_2(\mathbf{U})$ denote the weighted automaton obtained form $\mathbf{U}$ by projection over the output labels as described in Section 2. Then

$$\begin{aligned} \mathbf{Q}_i^\top \boldsymbol{\alpha} &= \mathrm{D}\left(y_i \mathbf{X}_i \circ \mathbf{T} \circ \mathbf{T}^{-1} \circ \mathbf{W}\right) = \mathrm{D}((y_i \mathbf{X}_i \circ \mathbf{T}) \circ (\mathbf{W} \circ \mathbf{T})^{-1}) \\ &= \mathrm{D}\left(\Pi_2(y_i \mathbf{X}_i \circ \mathbf{T}) \circ \Pi_2(\mathbf{W} \circ \mathbf{T})\right) = \mathrm{D}(\mathbf{\Phi}'_i \circ \mathbf{W}'), \end{aligned} \tag{3}$$

Table 1. First iteration of SVMRATIONALKERNELS on the dataset given Figure 4. The last line gives the values of $\boldsymbol{\alpha}$ and $\mathbf{W}'$ at the end of the iteration.

i	$\boldsymbol{\alpha}$	$\mathbf{W}'$	$\boldsymbol{\Phi}'_i \circ \mathbf{W}'$	$\mathrm{D}(\boldsymbol{\Phi}'_i \circ \mathbf{W}')$	α'_i
1	$(0, 0, 0)$	Fig. 5(a)	Fig. 6(a)	0	$\frac{1}{8}$
2	$(\frac{1}{8}, 0, 0)$	Fig. 5(b)	Fig. 6(b)	$\frac{3}{4}$	$\frac{1}{24}$
3	$(\frac{1}{8}, \frac{1}{24}, 0)$	Fig. 5(c)	Fig. 6(c)	$-\frac{23}{24}$	$\frac{47}{144}$
	$(\frac{1}{8}, \frac{1}{24}, \frac{47}{144})$	Fig. 5(d)			

where $\boldsymbol{\Phi}'_i = \mathit{\Pi}_2(y_i \mathbf{X}_i \circ \mathbf{T})$ and $\mathbf{W}' = \mathit{\Pi}_2(\mathbf{W} \circ \mathbf{T})$. $\boldsymbol{\Phi}'_i$, $i \in [1, m]$ can be precomputed and instead of $\mathbf{W}$, we can equivalently maintain $\mathbf{W}'$, with the following update rule:

$$\mathbf{W}' \leftarrow \mathbf{W}' + \mathit{\Delta}(\alpha_i)\boldsymbol{\Phi}'_i. \tag{4}$$

The gradient $\nabla(F)(\boldsymbol{\alpha}) = \mathbf{Q}\boldsymbol{\alpha} - \mathbf{1}$ can be expressed as follows

$$[\nabla(F)(\boldsymbol{\alpha})]_j = [\mathbf{Q}^\top \boldsymbol{\alpha} - \mathbf{1}]_j = \mathbf{Q}_j^\top \boldsymbol{\alpha} - 1 = \mathrm{D}(\boldsymbol{\Phi}'_j \circ \mathbf{W}') - 1.$$

The update rule for the main term $\mathrm{D}(\boldsymbol{\Phi}'_j \circ \mathbf{W}')$ can be written as

$$\mathrm{D}(\boldsymbol{\Phi}'_j \circ \mathbf{W}') \leftarrow \mathrm{D}(\boldsymbol{\Phi}'_j \circ \mathbf{W}') + \mathrm{D}(\boldsymbol{\Phi}'_j \circ \mathit{\Delta}\mathbf{W}').$$

Using (3) to compute the gradient and (4) to update $\mathbf{W}'$, we can generalize Algorithm SVMCOORDINATEDESCENT of Figure 2 and obtain Algorithm SVMRATIONALKERNELS of Figure 3. It follows from [16] that this algorithm converges at least linearly towards a global optimal solution. Moreover, the heuristics used by [10] and mentioned in the previous section can also be applied here to empirically improve the convergence rate of the algorithm. Table 1 shows the first iteration of SVMRATIONALKERNELS on the dataset given by Figure 4 when using a bigram kernel.

5 Implementation and Analysis

A key factor in analyzing the complexity of SVMRATIONALKERNELS is the choice of the data structure used to represent $\mathbf{W}'$. In order to simplify the analysis, we assume that the $\boldsymbol{\Phi}'_i$s, and thus $\mathbf{W}'$, are acyclic. This assumption holds for all rational kernels used in practice, however, it is not a requirement for the correctness of SVMRATIONALKERNELS. Given an acyclic weighted automaton $\mathbf{A}$, we denote by $l(\mathbf{A})$ the maximal length of an accepting path in $\mathbf{A}$ and by $n(\mathbf{A})$ the number of accepting paths in $\mathbf{A}$.

A straightforward choice follows directly from the definition of $\mathbf{W}'$. $\mathbf{W}'$ is represented as a non-deterministic weighted automaton, $\mathbf{W}' = \sum_{i=1}^{m} \alpha_i \boldsymbol{\Phi}'_i$, with a single initial state and m outgoing ϵ-transitions, where the weight of the ith transition is α_i and its destination state the initial state of $\boldsymbol{\Phi}'_i$. The size of this choice of $\mathbf{W}'$ is $|\mathbf{W}'| = m + \sum_{i=1}^{m} |\boldsymbol{\Phi}'_i|$. The benefit of this representation is that the update of $\boldsymbol{\alpha}$ using (4) can be performed in constant time since it requires modifying only the weight of

Table 2. Time complexity of each gradient computation and of each update of $\mathbf{W}'$ and the space complexity required for representing $\mathbf{W}'$ given for each type of representation of $\mathbf{W}'$

Representation of $\mathbf{W}'$	Time complexity (gradient)	(update)	Space complexity (for storing $\mathbf{W}'$)
naive ($\mathbf{W}'_n$)	$O(\lvert\mathbf{\Phi}'_i\rvert \sum_{i=1}^{m} \lvert\mathbf{\Phi}'_i\rvert)$	$O(1)$	$O(m)$
trie ($\mathbf{W}'_t$)	$O(n(\mathbf{\Phi}'_i) l(\mathbf{\Phi}'_i))$	$O(n(\mathbf{\Phi}'_i))$	$O(\lvert\mathbf{W}'_t\rvert)$
minimal automaton ($\mathbf{W}'_m$)	$O(\lvert\mathbf{\Phi}'_i \circ \mathbf{W}'_m\rvert)$	open	$O(\lvert\mathbf{W}'_m\rvert)$

one of the ϵ-transitions out of the initial state. However, the complexity of computing the gradient using (3) is in $O(|\mathbf{\Phi}'_j||\mathbf{W}'|) = O(|\mathbf{\Phi}'_j| \sum_{i=1}^{m} |\mathbf{\Phi}'_i|)$.

Representing $\mathbf{W}'$ as a deterministic weighted trie can lead to a simple update using (4). A *weighted trie* is a rooted tree where each edge is labeled and each node is weighted. During composition, each accepting path in $\mathbf{\Phi}'_i$ is matched with a distinct node in $\mathbf{W}'$. Thus, $n(\mathbf{\Phi}'_i)$ paths of $\mathbf{W}'$ are explored during composition. Since the length of each of these paths is at most $l(\mathbf{\Phi}'_i)$, this leads to a complexity in $O\left(n(\mathbf{\Phi}'_i) l(\mathbf{\Phi}'_i)\right)$ for computing $\mathbf{\Phi}'_i \circ \mathbf{W}'$ and thus for computing the gradient using (3). Since each accepting path in $\mathbf{\Phi}'_i$ corresponds to a distinct node in $\mathbf{W}'$, the weights of at most $n(\mathbf{\Phi}'_i)$ nodes of $\mathbf{W}'$ need to be updated. Thus, the complexity of an update of $\mathbf{W}'$ is $O\left(n(\mathbf{\Phi}'_i)\right)$.

The drawback of a trie representation is that it does not provide all of the sparsity benefits of a fully automata-based approach. A more space-efficient approach consists of representing $\mathbf{W}'$ as a minimal deterministic weighted automaton which can be substantially smaller, exponentially smaller in some cases, than the corresponding trie.

The complexity of computing the gradient using (3) is then in $O(|\mathbf{\Phi}'_i \circ \mathbf{W}'|)$ which is significantly less than the $O\left(n(\mathbf{\Phi}'_i) l(\mathbf{\Phi}'_i)\right)$ complexity of the trie representation. Performing the update of $\mathbf{W}'$ using (4) can be more costly though. With the straightforward approach of using the general union, weighted determinization and minimization algorithms [5], the complexity depends on the size of $\mathbf{W}'$. The cost of an update can thus sometimes become large. However, it is perhaps possible to design more efficient algorithms for augmenting a weighted automaton with a single string or even a set of strings represented by a deterministic automaton, while preserving determinism and minimality. The approach just described forms a strong motivation for the study and analysis of such non-trivial and probably sophisticated automata algorithms since it could lead to even more efficient updates of $\mathbf{W}'$ and overall speed-up of the SVMs training with rational kernels. We leave the study of this open question to the future. We note, however, that that analysis could benefit from existing algorithms in the unweighted case. Indeed, in the unweighted case, a number of efficient algorithms have been designed for incrementally adding a string to a minimal deterministic automaton while keeping the result minimal and deterministic [7, 3], and the complexity of each addition of a string using these algorithms is only linear in the length of the string added.

Table 2 summarizes the time and space requirements for each type of representation for $\mathbf{W}'$. In the case of an n-gram kernel of order k, $l(\mathbf{\Phi}'_i)$ is a constant k, $n(\mathbf{\Phi}'_i)$ is the number of distinct k-grams occurring in $\mathbf{x}_i$, $n(\mathbf{W}'_t)$ $(= n(\mathbf{W}'_m))$ the number of distinct k-grams occurring in the dataset, and $|\mathbf{W}'_t|$ the number of distinct n-grams of order less than or equal to k in the dataset.

Table 3. Time for training an SVM classifier using an SMO-like algorithm and SVMRATIONALKERNELS using a trie representation for $\mathbf{W}'$, and size of $\mathbf{W}'$ (number of transitions) when representing $\mathbf{W}'$ as a deterministic weighted trie and a minimal deterministic weighted automaton

Dataset	Kernel	SMO-like	New Algo.	trie	min. aut.
Reuters	4-gram	2m 18s	25s	66,331	34,785
(subset)	5-gram	3m 56s	30s	154,460	63,643
	6-gram	6m 16s	41s	283,856	103,459
	7-gram	9m 24s	1m 01s	452,881	157,390
	10-gram	25m 22s	1m 53s	1,151,217	413,878
	gappy 3-gram	10m 40s	1m 23s	103,353	66,650
	gappy 4-gram	58m 08s	7m 42s	1,213,281	411,939
Reuters	4-gram	618m 43s	16m 30s	242,570	106,640
(full)	5-gram	>2000m	23m 17s	787,514	237,783
	6-gram	>2000m	31m 22s	1,852,634	441,242
	7-gram	>2000m	37m 23s	3,570,741	727,743

6 Experiments

We used the Reuters-21578 dataset, a large data set convenient for our analysis and commonly used in experimental analyses of string kernels (`http://www.daviddlewis.com/resources/`). We refer by *full dataset* to the 12,902 news stories part of the ModeApte split. Since our goal is only to test speed (and not accuracy), we train on training and test sets combined. We also considered a subset of that dataset consisting of 466 news stories. We experimented both with n-gram kernels and gappy n-gram kernels with different n-gram orders. We trained binary SVM classification for the `acq` class using the following two algorithms: (a) the SMO-like algorithm of [8] implemented using LIBSVM [4] and modified to handle the on-demand computation of rational kernels; and (b) SVMRATIONALKERNELS implemented using a trie representation for $\mathbf{W}'$. Table 3 reports the training time observed using a dual-core 2.2 GHz AMD Opteron workstation with 16GB of RAM, excluding the pre-processing step which consists of computing $\mathbf{\Phi}'_i$ for each data point and that is common to both algorithms. To estimate the benefits of representing $\mathbf{W}'$ as a minimal automaton, we applied the weighted minimization algorithm to the tries output by SVMRATIONALKERNELS (after shifting the weights to the non-negative domain) and observed the resulting reduction in size. The results reported in Table 3 show that representing $\mathbf{W}'$ by a minimal deterministic automaton can lead to very significant savings in space and a substantial reduction of the training time with respect to the trie representation using an incremental addition of strings to $\mathbf{W}'$.

7 Conclusion

We presented novel techniques for large-scale training of SVMs when used with sequence kernels. We gave a detailed description of our algorithms and discussed different implementation choices, and presented an analysis of the resulting complexity. Our empirical results with large-scale data sets demonstrate dramatic reductions of the training time. Our software will be made publicly available through an open-source project.

Remarkably, our training algorithm for SVMs is entirely based on weighted automata algorithms and requires no specific solver.

References

1. Allauzen, C., Mohri, M., Talwalkar, A.: Sequence kernels for predicting protein essentiality. In: ICML 2008 (2008)
2. Bach, F.R., Jordan, M.I.: Kernel independent component analysis. JMLR 3, 1–48 (2002)
3. Carrosco, R.C., Forcada, M.L.: Incremental construction and maintenance of minimal finite-state automata. Computational Linguistics 28(2), 207–216 (2002)
4. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines (2001)
5. Cortes, C., Haffner, P., Mohri, M.: Rational Kernels: Theory and Algorithms. JMLR (2004)
6. Cortes, C., Vapnik, V.: Support-Vector Networks. Machine Learning 20(3) (1995)
7. Daciuk, J., Mihov, S., Watson, B.W., Watson, R.: Incremental construction of minimal acyclic finite state automata. Computational Linguistics 26(1), 3–16 (2000)
8. Fan, R.-E., Chen, P.-H., Lin, C.-J.: Working set selection using second order information for training SVM. JMLR 6, 1889–1918 (2005)
9. Fine, S., Scheinberg, K.: Efficient SVM training using low-rank kernel representations. Journal of Machine Learning Research 2, 243–264 (2002)
10. Hsieh, C.-J., Chang, K.-W., Lin, C.-J., Keerthi, S.S., Sundararajan, S.: A dual coordinate descent method for large-scale linear SVM. In: ICML, pp. 408–415 (2008)
11. Joachims, T.: Making large-scale SVM learning practical. In: Advances in Kernel Methods: Support Vector Learning. The MIT Press, Cambridge (1998)
12. Kuich, W., Salomaa, A.: Semirings, Automata, Languages. In: EATCS Monographs on Theoretical Computer Science, vol. 5. Springer, New York (1986)
13. Kumar, S., Mohri, M., Talwalkar, A.: On sampling-based approximate spectral decomposition. In: ICML (2009)
14. Leslie, C.S., Eskin, E., Noble, W.S.: The Spectrum Kernel: A String Kernel for SVM Protein Classification. In: Pacific Symposium on Biocomputing, pp. 566–575 (2002)
15. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. JMLR 2 (2002)
16. Luo, Z.Q., Tseng, P.: On the convergence of the coordinate descent method for convex differentiable minimization. J. of Optim. Theor. and Appl. 72(1), 7–35 (1992)
17. Mohri, M.: Weighted automata algorithms. In: Handbook of Weighted Automata, pp. 213–254. Springer, Heidelberg (2009)
18. Salomaa, A., Soittola, M.: Automata-Theoretic Aspects of Formal Power Series. Springer, Heidelberg (1978)
19. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge Univ. Press, Cambridge (2004)
20. Tsang, I.W., Kwok, J.T., Cheung, P.-M.: Core vector machines: Fast SVM training on very large data sets. JMLR 6, 363–392 (2005)
21. Williams, C.K.I., Seeger, M.: Using the Nyström method to speed up kernel machines. In: NIPS, pp. 682–688 (2000)

Filters for Efficient Composition of Weighted Finite-State Transducers

Cyril Allauzen, Michael Riley, and Johan Schalkwyk

Google Research, 76 Ninth Avenue, New York, NY 10011, USA
{allauzen,riley,johans}@google.com

Abstract. This paper describes a weighted finite-state transducer composition algorithm that generalizes the concept of the *composition filter* and presents various filters that process epsilon transitions, look-ahead along paths, and push forward labels along epsilon paths. These filters, either individually or in combination, make it possible to compose some transducers much more efficiently in time and space than otherwise possible. We present examples of this drawn, in part, from demanding speech-processing applications. The generalized composition algorithm and many of these filters have been included in *OpenFst*, an open-source weighted transducer library.

1 Introduction

The *composition* algorithm plays a central role in the use of weighted finite-state transducers. It is used, for example, to apply finite-state models to inputs and to combine cascaded models. The classical version of the composition algorithm, which simply matches transitions leaving paired input states, is easy to implement and often effective in practice. However, experience has shown that there are some transducers of practical importance that do not compose efficiently in this way. These cases typically create significant numbers of non-coaccessible composition states that waste time and space. For some problems, it is possible to find equivalent inputs that will compose more efficiently, but it is not always possible or desirable to do so. This has been especially an issue in natural language processing applications and led to special-purpose composition algorithms for use in speech recognition [5,6,10,14] and speech synthesis [2].

In this paper we generalize the composition algorithm, subsuming several of these specializations and others in an efficient way. The idea is to introduce a composition *filter*, applied at each composition state during the construction, that decides if composition is to continue. If we set out to create a general composition filter that blocks every non-coaccessible composition state for any input transducers, then we have only delegated the job of doing a full composition to the filter. Instead, we take the view that there are certain specific filters, tailored to particular but common cases, that are efficient to use, involving only a limited degree of look-ahead along paths. Composition itself is then parameterized to take one or more of these filters that are selected by the user to fit his problem.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 28–38, 2011.

Section 2 presents the generalized composition algorithm and defines several composition filters. Section 3 provides examples of these composition filters applied to practical problems. Section 4 briefly describes how these filters are used in *OpenFst* [3], an open-source weighted transducer library.

2 Composition Algorithm

2.1 Preliminaries

A semiring $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$ is ring that may lack negation. If $\otimes$ is commutative, we say that the semiring is *commutative*.

The *probability semiring* $(\mathbb{R}_+, +, \times, 0, 1)$ is used when the weights represent probabilities. The *log semiring* $(\mathbb{R} \cup \{\infty\}, \oplus_{\log}, +, \infty, 0)$, isomorphic to the probability semiring via the negative-log mapping, is often used in practice for numerical stability. The *tropical semiring* $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$, derived from the log semiring using the *Viterbi approximation*, is often used in shortest-path applications.

A *weighted finite-state transducer* $T = (\mathcal{A}, \mathcal{B}, Q, I, F, E, \lambda, \rho)$ over a semiring $\mathbb{K}$ is specified by a finite input alphabet $\mathcal{A}$, a finite output alphabet $\mathcal{B}$, a finite set of states Q, a set of initial states $I \subseteq Q$, a set of final states $F \subseteq Q$, a finite set of transitions $E \subseteq \overline{E} = Q \times (\mathcal{A} \cup \{\epsilon\}) \times (\mathcal{B} \cup \{\epsilon\}) \times \mathbb{K} \times Q$, an initial state weight assignment $\lambda : I \to \mathbb{K}$, and a final state weight assignment $\rho : F \to \mathbb{K}$. $E[q]$ denotes the set of transitions leaving state $q \in Q$.

Given a transition $e \in E$, $p[e]$ denotes its origin or previous state, $n[e]$ its destination or next state, $i[e]$ its input label, $o[e]$ its output label, and $w[e]$ its weight. A *path* $\pi = e_1 \cdots e_k$ is a sequence of consecutive transitions: $n[e_{i-1}] = p[e_i]$, $i = 2, \ldots, k$. The functions n, p, and w on transitions can be extended to paths by setting: $n[\pi] = n[e_k]$ and $p[\pi] = p[e_1]$ and by defining the weight of a path as the $\otimes$-product of the weights of its constituent transitions: $w[\pi] = w[e_1] \otimes \cdots \otimes w[e_k]$. A *string* is a sequence of labels; ϵ denotes the empty string.

The weight associated by T to any pair of input-output strings (x, y) is given by:

$$T(x, y) = \bigoplus_{\pi \in \cup_{q \in I,\, q' \in F} P(q, x, y, q')} \lambda[p[\pi]] \otimes w[\pi] \otimes \rho[n[\pi]], \tag{1}$$

where $P(q, x, y, q')$ denotes the set of paths from q to q' with input label $x \in \mathcal{A}^*$ and output label $y \in \mathcal{B}^*$.

We denote by $|T|_Q$ the number of states, $|T|_E$ the number of transitions, and $d(T)$ the maximum out-degree in T. The *size* of T is then $|T| = |T|_Q + |T|_E$.

2.2 Composition

Let $\mathbb{K}$ be a commutative semiring and let T_1 and T_2 be two weighted transducers defined over $\mathbb{K}$ such that the input alphabet $\mathcal{B}$ of T_2 coincides with the output alphabet of T_1. The result of the composition of T_1 and T_2 is a weighted transducer denoted by $T_1 \circ T_2$ and specified for all x, y by:

$$(T_1 \circ T_2)(x, y) = \bigoplus_{z \in \mathcal{B}^*} T_1(x, z) \otimes T_2(z, y). \tag{2}$$

```
WEIGHTED-COMPOSITION(T_1, T_2, Φ)
 1  Q ← I ← S ← I_1 × I_2 × {i_3}
 2  for each (q_1, q_2, i_3) ∈ I do
 3     λ(q_1, q_2, i_3) ← λ_1(q_1) ⊗ λ_2(q_2)
 4  while S ≠ ∅ do
 5     (q_1, q_2, q_3) ← HEAD(S)
 6     DEQUEUE(S)
 7     if (q_1, q_2, q_3) ∈ F_1 × F_2 × Q_3 and ρ_3(q_3) ≠ 0̄ then
 8        F ← F ∪ {(q_1, q_2, q_3)}
 9        ρ(q_1, q_2, q_3) ← ρ_1(q_1) ⊗ ρ_2(q_2) ⊗ ρ_3(q_3)
10     M ← {(e_1, e_2) ∈ E^L[q_1] × E^L[q_2] s.t. φ(e_1, e_2, q_3) = (e'_1, e'_2, q'_3) with q'_3 ≠⊥}
11     for each (e_1, e_2) ∈ M do
12        (e'_1, e'_2, q'_3) ← φ(e_1, e_2, q_3)
13        if (n[e'_1], n[e'_2], q'_3) ∉ Q then
14           Q ← Q ∪ {(n[e'_1], n[e'_2], q'_3)}
15           ENQUEUE(S, (n[e'_1], n[e'_2], q'_3))
16        E ← E ∪ {((q_1, q_2, q_3), i[e'_1], o[e'_2], w[e'_1] ⊗ w[e'_2], (n[e'_1], n[e'_2], q'_3))}
17  return T
```

Fig. 1. Pseudocode of the composition algorithm

Leaving aside transitions with ϵ inputs or outputs, the following rule specifies how to compute a transition of $T_1 \circ T_2$ from appropriate transitions of T_1 and T_2: (q_1, a, b, w_1, q'_1) and (q_2, b, c, w_2, q'_2) results in $((q_1, q_2), a, c, w_1 \otimes w_2, (q'_1, q'_2))$. A simple algorithm to compute the composition of two ϵ-free transducers, following the above rule, is given in [13].

More care is needed when T_1 has output ϵ labels or T_2 input ϵ labels. An output ϵ label in T_1 may be matched with an input ϵ label in T_2, following the above rule with ϵ labels treated as regular symbols. However, an output ϵ label may also be read in T_1 without matching any actual transition in T_2. This case can be handled by the above rule after adding self-loops at every state of T_2 labeled on the inner tape by a new symbol ϵ^L and on the outer tape by ϵ and allowing transitions labeled by ϵ and ϵ^L to match. Similar self-loops are added to T_1 for matching input ϵ labels on T_2. However, this approach can result in redundant ϵ-paths since an epsilon label can match in the two above ways. The redundant paths must be *filtered* out because they will produce incorrect results in non-idempotent semirings (like the log semiring).[1] We introduced the ϵ^L label to distinguish these two types of match in the filtering.

In [13], a *filter transducer* is introduced that is used with relabeling and the ϵ-free composition algorithm to correctly implement composition with ϵ labels. Our composition algorithm extends this by generalizing the *composition filter*.

Our algorithm takes as input two weighted transducers $T_1 = (\mathcal{A}, \mathcal{B}, Q_1, I_1, F_1, E_1, \lambda_1, \rho_1)$ and $T_2 = (\mathcal{B}, \mathcal{C}, Q_2, I_2, F_2, E_2, \lambda_2, \rho_2)$ over a semiring $\mathbb{K}$ and a composition filter $\Phi = (T_1, T_2, Q_3, i_3, \bot, \varphi, \rho_3)$, which has a set of filter states Q_3, a designated initial filter state i_3, a designated blocking filter state $\bot$, a transition filter $\varphi : E_1^L \times E_2^L \times Q_3 \to \overline{E}_1 \times \overline{E}_2 \times Q_3$ where $E_n^L = \bigcup_{q \in Q_n} E^L[q]$, $E^L[q_1] = E[q_1] \cup \{(q_1, \epsilon, \epsilon^L, \overline{1}, q_1)\}$ for each $q_1 \in Q_1$, $E^L[q_2] = E[q_2] \cup \{(q_2, \epsilon^L, \epsilon, \overline{1}, q_2)\}$ for each $q_2 \in Q_2$ and a final weight filter $\rho_3 : Q_3 \to \mathbb{K}$.

[1] Redundant ϵ-paths are also an issue in the unweighted case when testing for the ambiguity of finite automata [1].

We shall see that the filter can be used in composition to block the expansion of some states (by entering the $\perp$ state) and modify the transitions and final weights (useful for optimizations).

The states in the output of composition are identified with triples of a state from each of the two input transducers and one from the filter. In particular, the algorithm outputs a weighted finite-state transducer $T = (\mathcal{A}, \mathcal{C}, Q, I, F, E, \lambda, \rho)$ implementing the composition of T_1 and T_2 where $Q \subseteq Q_1 \times Q_2 \times Q_3$ and $I = I_1 \times I_2 \times \{i_3\}$.

Figure 1 gives the pseudocode of this algorithm. E and F are all initialized to the empty set and grown as needed. The algorithm uses a queue S containing the set of state triples of states yet to be examined. The queue discipline of S is arbitrary and does not affect the termination of the algorithm. The state set Q is initially the set of triples of initial states of the original transducers and filter, as is I and S, and the corresponding initial weights are computed (lines 1-3). Each time through the loop in lines 3-14, a new triple of states (q_1, q_2, q_3) is extracted from S (lines 5-6). The final weight of (q_1, q_2, q_3) is computed by $\otimes$-multiplying the final weights of q_1 and q_2 and the final filter weight when they are all final states (lines 8-9). Then, for each pair of transitions, the transition filter is first applied. If the new filter state is not the blocking state $\perp$ and a new transition is created from the filter-rewritten transitions (e'_1, e'_2) (line 16). If the destination state $(n[e'_1], n[e'_2], q'_3)$ has not been found previously, it is added to Q and inserted in S (lines 13-15). The composition algorithm presented here is available in the *OpenFst* library [3].

2.3 Elementary Composition Filters

In this section, we consider elementary filters for composition without and with epsilon transitions.

Trivial Filter. Filter Φ_{trivial} blocks no paths and leaves transitions and final weights unmodified. For Φ_{trivial}, let $Q_3 = \{0, \perp\}$, $i_3 = 0$, $\varphi(e_1, e_2, q_3) = (e_1, e_2, q'_3)$ with $q'_3 = 0$ if $o[e_1] = i[e_2] \in \mathcal{B}$ and $\perp$ otherwise, and $\rho(q_3) = \overline{1}$ for all $q_3 \in Q_3$. With this filter, the pseudocode in Figure 1 matches the simple epsilon-free composition algorithm given in [13].

Let us assume that the transitions at each state in T_2 are sorted according to their input label. The set M of transitions to be computed line 8 is simply equal to $\{(e_1, e_2) \in E[q_1] \times E[q_2] : o[e_1] = i[e_2]\}$. It can be computed by performing a binary search over $E[q_2]$ for each transition in $E[q_1]$. The time complexity of computing M is then $O(|E[q_1]| \log |E[q_2]| + |M|)$. Since each element in M will result in a transition in T, the worst-case time complexity of the algorithm is $O(|T|_Q d(T_1) \log d(T_2) + |T|_E)$. The space complexity of the algorithm is $O(|T|)$.

Epsilon-Matching Filter. Filter $\Phi_{\epsilon\text{-match}}$ handles epsilon labels, but disallows redundant epsilon paths, preferring those that match actual ϵ labels. It leaves transitions and final weights unmodified.

For $\Phi_{\epsilon\text{-match}}$, let $Q_3 = \{0, 1, 2, \bot\}$, $i_3 = 0$, $\rho(q_3) = \overline{1}$ for all $q_3 \in Q_3$, and $\varphi(e_1, e_2, q_3) = (e_1, e_2, q_3')$ where:

$$q_3' = \begin{cases} 0 & \text{if } (o[e_1], i[e_2]) = (x, x) \text{ with } x \in \mathcal{B}, \\ 0 & \text{if } (o[e_1], i[e_2]) = (\epsilon, \epsilon) \text{ and } q_3 = 0, \\ 1 & \text{if } (o[e_1], i[e_2]) = (\epsilon^L, \epsilon) \text{ and } q_3 \neq 2, \\ 2 & \text{if } (o[e_1], i[e_2]) = (\epsilon, \epsilon^L) \text{ and } q_3 \neq 1, \\ \bot & \text{otherwise.} \end{cases}$$

With this filter, the pseudocode in Figure 1 matches the composition algorithm given in [13] with the specified composition filter transducer. The complexity of the algorithm is the same as when using the trivial filter.

Epsilon-Sequencing Filter. Alternatively, filter $\Phi_{\epsilon\text{-seq}}$ can also be used to remove redundant epsilon paths. This filter favors epsilon paths consisting of (output) ϵ-transitions in T_1 (matched with staying at the same state in T_2) followed by (input) ϵ-transitions in T_2 (matched with staying at the same state in T_1).

For $\Phi_{\epsilon\text{-seq}}$, let $Q_3 = \{0, 1, \bot\}$, $i_3 = 0$, $\rho(q_3) = \overline{1}$ for all $q_3 \in Q_3$, and $\varphi(e_1, e_2, q_3) = (e_1, e_2, q_3')$ where:

$$q_3' = \begin{cases} 0 & \text{if } (o[e_1], i[e_2]) = (x, x) \text{ with } x \in \mathcal{B}, \\ 0 & \text{if } (o[e_1], i[e_2]) = (\epsilon, \epsilon^L) \text{ and } q_3 = 0, \\ 1 & \text{if } (o[e_1], i[e_2]) = (\epsilon^L, \epsilon), \\ \bot & \text{otherwise.} \end{cases} \tag{3}$$

The complexity of the algorithm is the same as when using the trivial filter. Replacing the pair $(o[e_1], i[e_2])$ by $(i[e_2], o[e_1])$ in (3) leads to the symmetric filter $\overline{\Phi}_{\epsilon\text{-seq}}$. Whether it is better to choose the epsilon-matching or epsilon-sequencing filter is problem-dependent as shown in Section 3.

2.4 Look-Ahead Composition Filters

In this section, we introduce filters that can result in more efficient composition by looking-ahead along paths and blocking unsuccessful matches under various scenarios.

String-Potential Filter. Filter Φ_{sp} looks-ahead along common prefixes of state futures. Given two strings u and v, we denote by $u \wedge v$ the longest common prefix of u and v. Given a state q in a tranducer T, the input (resp. output) string potential of q, denoted by $p_i(q)$ (resp. $p_o(q)$), is the longest common prefix of the input (resp. output) labels of all the paths from q to a final state.

For Φ_{sp}, let $Q_3 = \{0, \bot\}$, $i_3 = 0$, $\rho(0) = \overline{1}$, and $\varphi(e_1, e_2, q_3) = (e_1, e_2, q_3')$ where:

$$q_3' = \begin{cases} 0 & \text{if } p_o(n[e_1]) \wedge p_i(n[e_2]) \in \{p_o(n[e_1]), p_i(n[e_2])\}, \\ \bot & \text{otherwise.} \end{cases}$$

This filter prevents the creation of some non-coaccessible states since a state (q_1, q_2) in $T_1 \circ T_2$ is coaccessible only if $p_o(q_1)$ is a prefix of $p_i(q_2)$ or $p_i(q_2)$ is a

prefix of $p_o(q_1)$ [2]. Computing string potentials can be done using the generic single-source shortest-distance algorithm of [12] over the string semiring. This can be done on-demand or as a pre-processing step. Naively storing a string at each state results in a complexity (on-demand) of $O(|T|_Q d(T_1) \log d(T_2) + |T|_E \min(\mu_1, \mu_2))$ in time and $O(|T| + |T_1|_Q \mu_1 + |T_2|_Q \mu_2)$ in space, with μ_i being the length of the longest potential in T_i. This can be improved using better data structures (such as tries or suffix trees).

Transition-Look-Ahead Filter. When states paired in composition have no shared common prefixes, it is is necessary to examine the specific transitions themselves in any look-ahead. A simple form of look-ahead is then to try to match one set of transitions into the future.

Given a state q in a transducer T let us denote by $L_i(q)$ and $L_o(q)$ the set of input and output labels of outgoing transitions in q. For $\Phi_{\text{tr-la}}$, let $Q_3 = \{0, \bot\}$, $i_3 = 0$, $\rho(0) = \overline{1}$, and $\varphi(e_1, e_2, q_3) = (e_1, e_2, q'_3)$ where:

$$q'_3 = \begin{cases} 0 & \text{if } L_o(n[e_1]) \cap L_i(n[e_2]) \neq \emptyset \text{ or } \epsilon \in L_o(n[e_1]) \cup L_i(n[e_2]), \\ \bot & \text{otherwise.} \end{cases}$$

The sets $L_i(q)$ and $L_o(q)$ can be computed on-demand or as a pre-processing step and can be represented using data-structures providing efficient intersection such as bit vectors or Bloom filters. Using bit vectors, the complexity (on-demand) is $O(|T|_Q d(T_1) \log d(T_2) + |T|_E \log |\mathcal{B}|)$ in time and $O(|T| + (|T_1|_Q + |T_2|_Q) \log |\mathcal{B}|)$ in space.

Label-Reachability Filter. In transducers with epsilon transitions, looking-ahead a single transition is not sufficient, since we can not match a (non-epsilon) label without traversing epsilon paths. Filter Φ_{reach} precomputes those traverals.

When composing states q_1 in T_1 and q_2 in T_2, filter Φ_{reach} disallows following an epsilon-labeled path from q_1 that will fail to reach a non-epsilon label that matches some transition leaving state q_2. It leaves transitions and final weights unmodified. For simplicity, we assume there are no input ϵ labels in T_1.

For Φ_{reach}, let $Q_3 = \{0, \bot\}$, $i_3 = 0$, and $\rho(q_3) = \overline{1}$ for all $q_3 \in Q_3$. Define $r : \mathcal{B} \times Q_1 \to \{0, 1\}$ such that $r(x, q) = 1$ if there is a path π from q to some q' in T_1 with $o[\pi] = x$, otherwise let $r(x, q) = 0$. Let $\varphi(e_1, e_2, q_3) = (e_1, e_2, 0)$ if (i) $o[e_1] = i[e_2]$ or if (ii) $o[e_1] = \epsilon, i[e_2] = \epsilon^L$, and for some $e'_2 \in E[p[e_2]]$, $i[e'_2] \neq \epsilon$ and $r(i[e'_2], n[e_1]) = 1$. Otherwise let $\varphi(e_1, e_2, q_3) = (e_1, e_2, \bot)$.

Let us denote by $c_r(T_1)$ the cost of performing one reachability query in T_1 using r, by $S_r(T_1)$ the total space required for r, and by $d_\epsilon T_1$ the maximal number of output-ϵ transitions at a state in T_1. The worst-case time complexity of the algorithm is: $O(|T|_Q (d(T_1) \log d(T_2) + d_\epsilon(T_1) c_r(T_1)) + |T|_E)$, and the space complexity is $O(|T| + S_r(T_1))$.

There are different ways we can represent r and they will lead to different complexities for composition. We will assume for our analysis, whatever its representation, that r is precomputed and stored with T_1. In general, we exclude any T-specific precomputation from composition's time complexity.

Point Representation of r: Define $R_q = \{x \in \mathcal{B} : r(x,q) = 1\}$ for each state $q \in T_1$. If the labels in R_q are stored in a linked list, traversed linearly and each matched against sorted input labels in T_2 using binary search, then $c_r(T_1) = \max_q |R_q| \log d(T_2)$ and $S_r(T_1) = \sum_q |R_q|$.

Interval Representation of r: We can use intervals to represent R_q if $\mathcal{B} = [1, |\mathcal{B}|] \subset \mathbb{N}$ by defining $I_q = \{[x,y) : x, y \in \mathbb{N}, [x,y) \subseteq R_q, x-1 \notin R_q, y \notin R_q\}$. If the intervals in I_q are stored in a linked list, traversed linearly and each matched against sorted input labels in T_2 using (lower-bound) binary search, then $c_r(T_1) = \max_q |I_q| \log d(T_2)$ and $S_r(T_1) = \sum_q |I_q|$.

Assuming the particular numbering of the labels is arbitrary, let permutation $\Pi : \mathcal{B} \to \mathcal{B}$ be a bijection that is used to relabel both T_1 and T_2 prior to composition. Among the $|\mathcal{B}|!$ different possible such permutations, some could result in far fewer intervals in I_q than others. In fact, there may exist a Π that results in one interval per I_q. Consider the $|\mathcal{B}| \times |Q_1|$ matrix $\mathbf{R}$ with $\mathbf{R}[i,j] = r(i,j)$. The condition that the I_q each contain a single interval is equivalent to the property that the ones in the columns of $\mathbf{R}$ are consecutive. A binary matrix $\mathbf{R}$ that has a permutation of rows that results in columns with consecutive ones is said to have the *Consecutive One's Property* (C1P). The problem has been extensively studied and has many applications [4,8,9,11]. There are linear algorithms to find a permutation if it exists; the first, due to Booth and Lucker, was based on PQ-trees [4]. There are approximate algorithms when an exact solution does not exist [7]. Our speech application that follows admits C1P. As such, the interval representation of r results in a significant complexity reduction over the point representation.

Label-Reachability Filter with Label Pushing. A modification of the label-reachability filter for the case of a single transition matching leads to smaller and more efficient compositions as we will show in Section 3.

When matching an ϵ-transition e_1 in q_1 with an ϵ^L-loop in q_2, the Φ_{reach} filter allows this match if and only the set of transitions in q_2 that match the future in $n[e_1]$ is non-empty. In the special case where this set contains a unique transition e_2', the $\Phi_{\text{push-label}}$ filter allows e_1 to match e_2', resulting in the early output of $o[e_2']$.

For $\Phi_{\text{push-label}}$, let $Q_3 = \{\epsilon, \perp\} \cup \mathcal{B}$, $i_3 = \epsilon$ and $\rho(q_3) = \overline{1}$ if $q_3 = \epsilon$ and $\rho(q_3) = \overline{0}$ otherwise. Let $\varphi(e_1, e_2, q_3) = (e_1, e_2, \epsilon)$ if $q_3 = \epsilon$ and $o[e_1] = i[e_2]$, or if $q_3 = o[e_1] = \epsilon$, $i[e_2] = \epsilon^L$ and $|\{e \in E[q_2] : r(n[e_1], i[e]) = 1\}| \geq 2$, or if $q_3 = o[e_1] \neq \epsilon$ and $i[e_2] = \epsilon^L$. Let $\varphi(e_1, e_2, q_3) = (e_1, e_2, q_3)$ if $q_3 \neq \epsilon$, $o[e_1] = \epsilon$, $i[e_2] = \epsilon^L$ and $r(n[e_1], q_3) = 1$. Let $\varphi(e_1, e_2, \epsilon) = (e_1, e_2', i[e_2'])$ if $o[e_1] = \epsilon$, $i[e_2] = \epsilon^L$ and $\{e \in E[q_2] : r(n[e_1], i[e]) = 1\} = \{e_2'\}$. Otherwise, let $\varphi(e_1, e_2, q_3) = (e_1, e_2, \perp)$.

The complexity of the algorithm is the same as when using the label-reachability filter.

2.5 Combining Filters

In Section 2.3 we presented composition filters for correctly handling epsilon transitions and in Section 2.4 we presented look-ahead filters that can lead to

more efficient composition. In practice, we may need a combination of these filters, for example, to match with epsilon transitions and look-ahead along paths in a particular way. We present here how to synthesize a new composition filter from two components filters.

Let $\Phi^a = (Q_3^a, i_3^a, \perp^a, \varphi^a, \rho_3^a)$ and $\Phi^b = (Q_3^b, i_3^b, \perp^b, \varphi^b, \rho_3^b)$ be two composition filters, we will define their combination as the filter $\Phi^a \diamond \Phi^b = (Q_3, i_3, \perp, \varphi, \rho_3)$ with $Q_3 = Q_3^a \times Q_3^b$, $i_3 = (i_3^a, i_3^b)$, $\perp = (\perp^a, \perp^b)$, $\rho_3((q_3^a, q_3^b)) = \rho_3^a(q_3^a) \otimes \rho_3^b(q_3^b)$, and with φ defined as follows: given $(e_1, e_2, q_3) \in E_1 \times E_2 \times Q_3$ with $q_3 = (q_3^a, q_3^b)$, $\varphi^b(e_1, e_2, q_3^b) = (e_1', e_2', r_3^b)$ and $\varphi^a(e_1', e_2', q_3^a) = (e_1'', e_2'', r_3^a)$, then let

$$\varphi(e_1, e_2, q_3) = (e_1'', e_2'', q_3') \text{ with } q_3' = \begin{cases} \perp & \text{if } r_3^a = \perp^a \text{ or } r_3^b = \perp^b, \\ (r_3^a, r_3^b) & \text{otherwise.} \end{cases}$$

The filter $\Phi_{\text{reach}} \diamond \overline{\Phi}_{\epsilon\text{-seq}}$ can for instance be used to benefit from the label-reachable filter when T_2 contains input ϵ-transitions.

3 Examples

In this section, examples are given of the previously-defined composition filters. All examples are benchmarked using the composition algorithm in *OpenFst* [3].

Let $\Sigma = \{1, \ldots, 5000\}$ and let D be the two-state transducer over $\Sigma \times \Sigma$ that transduces each input symbol to ϵ as depicted in Figure 2(a). Consider the composition $D \circ D^{-1}$ using the epsilon-matching and epsilon-sequencing filters. The former creates a two-state machine with a transition for every element of $\Sigma \times \Sigma$ while the latter is identical to the concatenation TT^{-1}. Table 1(a)-(b) compares the number of composition states, transitions, time and memory usage with these two filters. In this example, the epsilon-sequencing filter gives a much smaller and efficiently-generated result than the epsilon-matching filter. It is easy to find examples where the opposite is true.

For the look-ahead filters, we draw our examples from a standard large-vocabulary speech recognition task - DARPA Broadcast News (BN). There are three alphabets for this task: Ω, the set of BN English words used where $|\Omega| = 70{,}897$; Π, the set of English phonemes where $|\Pi| = 46$; and Υ, a set of English tri-phonemic acoustic models where $|\Upsilon| = 20{,}910$. There are three component transducers for this task:

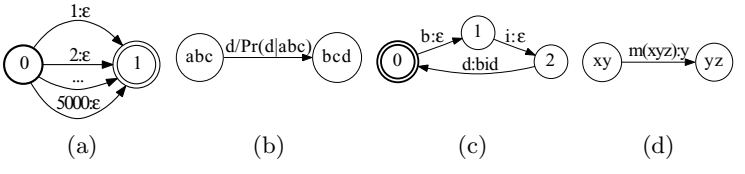


Fig. 2. Example transducers: (a) deleting transducer D, (b) n-gram language model G transition, (c) pronunciation lexicon L path, and (d) context-dependency transducer C transition

Table 1. Number of composition states and transitions (before trimming), time and memory usage for various composition filters. Observe that (a), (c), (e) and (g) correspond to using the composition algorithm from [13]. Experiments were conducted on a quad-core 2.2 GHz AMD Opteron machine with 32 GB of RAM.

	composition filter	T_1	T_2	$T_1 \circ T_2$ states	$T_1 \circ T_2$ transitions	time (sec)	mem. (mbytes)
(a)	`epsilon-matching`	D	D^{-1}	2	25,000,000	4.21	1419.5
(b)	`epsilon-sequencing`	D	D^{-1}	3	10,000	0.73	22.0
(c)	`trivial`	C	α	47,021,923	47,021,922	48.45	4704.0
(d)	`string-potential`	C	α	1,043,734	1,043,733	8.97	351.0
(e)	`trivial`	C	L	1,952,555	3,527,612	2.77	225.0
(f)	`transition-look-ahead`	C	L	120,489	149,972	0.84	33.4
(g)	`epsilon-sequencing`	L	G	?	?	$>$ 7200.00	$>$ 32,768.0
(h)	`label-reachability`	L	G	30,884,222	39,965,633	177.93	3612.9
(i)	`lab.-reach. w/ label-pushing`	L	G	13,377,323	22,151,870	113.72	1885.9

- a 4-gram *language model* G, which is a weighted automaton over Ω and has 2,213,539 states and 10,225,015 transitions. The weights model the probability of a particular sentence being uttered as estimated from the BN corpus. Figure 2(b) depicts the 4-gram transition *abcd* in G with probablity $Pr(d|abc)$.
- a minimal deterministic *lexicon transducer* L over $\Omega \times \Pi$, which maps phonemic pronunications to their word symbols and has 63,283 states and 145,710 transitions. The pronunciations are from a pronunciation dictionary. Figure 2(c) depicts a path in L.
- a minimal deterministic tri-phonemic *context-dependency transducer* C over $\Upsilon \times \Pi$, which maps from tri-phonemic model sequences to their corresponding phonemic sequence and has 1454 states and 88,840 transitions. The acoustic models are produced in the acoustic training phase of speech recognition and model a phoneme in its left and right context (possibly clustered due to data sparsity). Figure 2(d) depicts the transition in C for the triphonemic *xyz* model, *m(xyz)*.

For precise details about their form and construction of these three transducers, see [13]. We have chosen these transducers since the composition $C \circ L \circ G$, mapping from tri-phonemic models to word sequences weighted by their probabilities, is the *recognition transducer* matched against acoustic input during the recognition of an utterance. However, both C and L present significant issues for classical composition as detailed below. By constructing C and L differently, it is possible to use classical composition more efficiently, however these constructions introduce considerable non-determinism in the result that requires an expensive determinization to remove, something that we often wish to avoid.

While these examples are drawn from speech recognition, other application areas (e.g. text-to-speech synthesis, optical character recognition, spelling correction) involve similar language models, dictionaries and/or context-dependent constraints that can be modeled usefully with transducers and present similar issues with composition.

In the examples below that involve ϵ-transitions, we in fact use look-ahead filters combined with the epsilon-sequencing filter as described in Section 2.5.

String-Potential Filter: As depicted in Figure 2(d), a single symbol (the right tri-phoneme) is the output label for each transition leaving a state in the C transducer. That symbol is also the string potential at each state. In composition, we can take advantage of this as demonstrated by Table 1(c)-(d), which compares C composed with a random string $\alpha \in \Pi^{1000000}$ using the trivial versus the string-potential filters. The trivial filter is inefficient due to the output non-determinism, while the string-potential filter is much better in both time and space. Another effective use of string potentials in composition is given in [2].

Transition-Look-Ahead Filter: Unlike the previous example, the composition $C \circ L$ will not benefit much from using the string-potential filter since the string potential at most states in L is ϵ. In this case, the transition-look-ahead filter can be applied. Table 1(e)-(f), which compares the trivial and transition-look-ahead filters, demonstrates that the transition-look-ahead filter creates fewer states in the (untrimmed) result, saving time and space.

Label-Reachability Filter: The composition $L \circ G$ using the epsilon-sequencing (or -matching) composition filter is very inefficient since the initial epsilon paths in L create many non-coaccessible states in the result. For this problem, the label-reachability filter is appropriate. Table 1(g)-(h) compares the epsilon-sequencing and label-reachability filters. With the epsilon-sequencing filter, composition terminates after 2 hours with RAM exhausted, while with the label-reachability filter, only a few minutes are needed for completion.

Label-Reachability Filter with Label Pushing: While the label-reachability filter addresses the non-coaccessible states in the composition $L \circ G$ (in fact, the result is trim), it can further benefit from including label-pushing in the filter. Table 1(i) shows that if we do so, the result is smaller, builds faster and uses less memory. This benefit is due, in part, to all transitions entering a state in G having the same label.

4 Implementation

In *OpenFst* [3], the default composition filter is the epsilon-sequencing filter. It can be easily and very efficiently changed via templated options. For example, to use the epsilon-matching filter, one invokes:

```
ComposeFstOptions<StdArc, MatchComposeFilter> opts;
ComposeFst<StdArc> result(t1, t2, opts);
```

All filters described here are available in *OpenFst*. Further, users can add new ones by creating a class that meets the composition filter interface to handle their specific applications.

Acknowledgements. We thank Mehryar Mohri for suggesting using a generalized composition filter for solving problems such as those addressed here.

References

1. Allauzen, C., Mohri, M., Rastogi, A.: General algorithms for testing the ambiguity of finite automata. In: Ito, M., Toyama, M. (eds.) DLT 2008. LNCS, vol. 5257, pp. 108–120. Springer, Heidelberg (2008)
2. Allauzen, C., Mohri, M., Riley, M.: Statistical modeling for unit selection in speech synthesis. In: Proc. ACL, pp. 55–62 (2004)
3. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFst: A general and efficient weighted finite-state transducer library. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 11–23. Springer, Heidelberg (2007), http://www.openfst.org
4. Booth, K., Lueker, G.: Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. J. of Computer and System Sci. 13, 335–379 (1976)
5. Caseiro, D., Trancoso, I.: A specialized on-the-fly algorithm for lexicon and language model composition. IEEE Trans. on Audio, Speech and Lang. Proc. 14(4), 1281–1291 (2006)
6. Cheng, O., Dines, J., Doss, M.: A generalized dynamic composition algorithm of weighted finite state transducers for large vocabulary speech recognition. In: Proc. ICASSP, vol. 4, pp. 345–348 (2007)
7. Dom, M., Niedermeier, R.: The search for consecutive ones submatrices: Faster and more general. In: Proc. ACID, pp. 43–54 (2007)
8. Habib, M., McConnell, R., Paul, C., Viennot, L.: Lex-BFS and partition refinement with applications to transitive orientation, interval graph recognition and consecutive ones testing. Theor. Comput. Sci. 234, 59–84 (2000)
9. Hsu, W.-L., McConnell, R.: PC trees and circular-ones arrangements. Theor. Comput. Sci. 296(1), 99–116 (2003)
10. McDonough, J., Stoimenov, E., Klakow, D.: An algorithm for fast composition of weighted finite-state transducers. In: Proc. ASRU (2007)
11. Meidanis, J., Porto, O., Telles, G.: On the consecutive ones property. Discrete Appl. Math. 88, 325–354 (1998)
12. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics 7(3), 321–350 (2002)
13. Mohri, M., Pereira, F., Riley, M.: Speech recognition with weighted finite-state transducers. In: Jacob Benesty, Y.H., Sondhi, M. (eds.) Handbook of Speech Processing, pp. 559–582. Springer, Heidelberg (2008)
14. Oonishi, T., Dixon, P., Iwano, K., Furui, S.: Implementation and evaluation of fast on-the-fly WFST composition algorithms. In: Proc. Interspeech, pp. 2110–2113 (2008)

Incremental DFA Minimisation*

Marco Almeida**, Nelma Moreira, and Rogério Reis

DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal
{mfa,nam,rvr}@ncc.up.pt

Abstract. We present a new incremental algorithm for minimising deterministic finite automata. It runs in quadratic time for any practical application and may be halted at any point, returning a partially minimised automaton. Hence, the algorithm may be applied to a given automaton at the same time as it is processing a string for acceptance. We also include some experimental comparative results.

1 Introduction

We present a new algorithm for incrementally minimise deterministic finite automata. This algorithm may be halted at any point, returning a partially minimised automaton that recognises the same language as the input. Should the minimisation process be interrupted, calling the incremental minimisation algorithm with the output of the halted process would resume the minimisation process. Moreover, the algorithm can be run on some automaton D at the same time as D is being used to process a string for acceptance.

Unlike the usual approach, which computes the equivalence classes of the set of states, this algorithm proceeds by testing the equivalence of pairs of states in the same line of Watson and Daciuk [Wat01, WD03]. The intermediate results are stored for the speedup of ulterior computations in order to assure quadratic running time and memory usage.

This paper is structured as follows. In the next Section some basic concepts and notation are introduced. Section 3 is a small survey of related work. In Section 4 we describe the new algorithm in detail, presenting the proofs of correctness and worst-case running-time complexity. Section 5 follows with experimental comparative results, and Section 6 finishes with some conclusions and future work.

2 Preliminaries

We recall here the basic definitions needed throughout the paper. For further details we refer the reader to the work of Hopcroft et al. [HMU00].

* This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI, project ASA (PTDC/MAT/65481/2006) through Programs COMPETE and FEDER, and project CANTE (PTDC/EIA-CCO/101904/2008).
** Marco Almeida is funded by FCT grant SFRH/BD/27726/2006.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 39–48, 2011.

An alphabet Σ is a nonempty set of symbols. A word over an alphabet Σ is a finite sequence of symbols of Σ. The empty word is denoted by ϵ and the length of a word w is denoted by $|w|$. The set $\Sigma^\star$ is the set of words over Σ. A language L is a subset of $\Sigma^\star$.

A *deterministic finite automaton* (DFA) is a tuple $D = (Q, \Sigma, \delta, q_0, F)$ where Q is finite set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ the transition function, q_0 the initial state, and $F \subseteq Q$ the set of final states. We can extend the transition function to words $w \in \Sigma^\star$ such that $w = au$ by considering $\delta(q, w) = \delta(\delta(q, a), u)$ for $q \in Q$, $a \in \Sigma$, and $u \in \Sigma^\star$. The *language* accepted by the DFA D is $L(D) = \{w \in \Sigma^\star \mid \delta(q_0, w) \in F\}$. Two finite automata A and B are *equivalent*, denoted by $A \sim B$, if they accept the same language. For any DFA $D = (Q, \Sigma, \delta, q_0, F)$, let $\varepsilon(q) = 1$ if $q \in F$ and $\varepsilon(q) = 0$ otherwise, for $q \in Q$. Two states $q_1, q_2 \in Q$ are said to be *equivalent*, denoted by $q_1 \sim q_2$, if for every $w \in \Sigma^\star$, $\varepsilon(\delta(q_1, w)) = \varepsilon(\delta(q_2, w))$. A DFA is *minimal* if there is no equivalent DFA with fewer states. Given a DFA D, the equivalent minimal DFA $D/_\sim$ is called the *quotient automaton* of D by the equivalence relation $\sim$. Minimal DFAs are unique up to isomorphism.

2.1 The UNION-FIND Algorithm

The UNION-FIND [Tar75, CLRS03] algorithm takes a collection of n distinct elements grouped into several disjoint sets and performs two operations on it: merges two sets and finds to which set a given element belongs to. The algorithm is composed by the following three functions:

- MAKE(i): creates a new set (singleton) for one element i (the identifier);
- FIND(i): returns the identifier S_i of the set that contains i;
- UNION(i, j, k): combines the sets identified by i and j in a new set $S_k = S_i \cup S_j$; S_i and S_j are destroyed.

An important detail of the UNION operation is that the two combined sets are destroyed in the end. Our implementation of the algorithm (using rooted trees) follows the one by Cormen et al. [CLRS03]. The main claim is that an arbitrary sequence of i MAKE, UNION, and FIND operations, j of which are MAKE, can be performed in $O(i\alpha(j))$, where $\alpha(j)$ is related to a functional inverse of the Ackermann function, and, as such, grows *very* slowly. In fact, for every *practical* values of j (up to 16^{512}), $\alpha(j) \leq 4$.

3 Related Work

The problem of writing efficient algorithms to find the minimal equivalent DFA can be traced back to the 1950's with the works of Huffman [Huf55] and Moore [Moo58]. Over the years several alternative algorithms were proposed. In terms of worst-case complexity the best know algorithm (log-linear) is by Hopcroft [Hop71]. Brzozowski [Brz63] presented an elegant but exponential algorithm that may also be applied to non-deterministic finite automata.

The first DFA incremental minimisation algorithm was proposed by Watson [Wat01]. The worst-case running-time complexity of this algorithm is exponential — $O(k^{\max(0,n-2)})$, for a DFA with n states over an alphabet of k symbols. As shown by Watson himself [Wat95], this bound is tight. Later, Watson and Daciuk [WD03] proposed a new version of the algorithm. By using a memoization technique they achieved an almost quadratic run-time. Recently, however, a bug was found on the algorithm and one of the authors is currently trying to fix it.

4 The Incremental Minimisation Algorithm

Given an arbitrary DFA D as input, this algorithm may be halted at any time returning a partially minimised DFA that has no more states than D and recognises the same language. It uses a disjoint-set data structure to represent the DFA's states and the UNION-FIND algorithm to keep and update the equivalence classes. This approach allows us to maintain the transitive closure in a very concise and elegant manner. The pairs of states already marked as distinguishable are stored in an auxiliary data structure in order to avoid repeated computations.

Let $D = (Q, \Sigma, \delta, q, F)$ be a DFA with $n = |Q|$ and $k = |\Sigma|$. We assume that the states are represented by integers, and thus it is possible to order them. This ordering is used to normalise pairs of states, as presented in Listing 1.1.

```
1  def NORMALISE(p, q):
2      if p < q:
3          pair = (p, q)
4      else:
5          pair = (q, p)
6      return pair
```

Listing 1.1. A simple normalisation step

The normalisation step allows us to improve the behaviour of the minimisation algorithm by ensuring that only $\frac{n^2}{2} - n$ pairs of states are considered.

The quadratic time bound of the minimisation procedure MIN-INCR, presented in Listing 1.2, is achieved by testing each pair of states for equivalence exactly once. We assure this by storing the intermediate results of all calls to the pairwise equivalence-testing function EQUIV-P, defined in Listing 1.3. Some auxiliary data structures, designed specifically to improve the worst-case running time, are presented in Listing 1.4.

```
1  def MIN-INCR(D = (Q, Σ, δ, q0, F)):
2      for q ∈ Q:
3          MAKE(q)
4      NEQ = {NORMALISE(p, q) | p ∈ F, q ∈ Q − F}
5      for p ∈ Q:
6          for q ∈ {x | x ∈ Q, x > p}:
7              if (p, q) ∈ NEQ:
8                  continue
9              if FIND(p) = FIND(q):
10                 continue
```

```
11                  EQUIV = SET-MAKE(|Q|^2)
12                  PATH = SET-MAKE(|Q|^2)
13                  if EQUIV-P(p, q):
14                      for (p', q') ∈ SET-ELEMENTS(EQUIV):
15                          UNION(p', q')
16                  else:
17                      for (p', q') ∈ SET-ELEMENTS(PATH):
18                          NEQ = NEQ ∪ {(p', q')}
19      classes = {}
20      for p ∈ Q:
21          lider = FIND(p)
22          classes[lider] = classes[lider] ∪ {p}
23      D' = D
24      joinStates(D', classes)
25      return D'
```

Listing 1.2. Incremental DFA minimisation in quadratic time

Algorithm MIN-INCR starts by creating the initial equivalence classes (lines 2–3); these are singletons as no states are yet marked as equivalent. The global variable `NEQ`, used to store the distinguishable pairs of states, is also initialised (line 4) with the trivial identifications. Variables `PATH` and `EQUIV`, also global and reset before each call to EQUIV-P, maintain the history of calls to the transition function and the set of potentially equivalent pairs of states, respectively.

The main loop of MIN-INCR (lines 5–18) iterates through all the normalised pairs of states and, for those not yet known to be either distinguishable or equivalent, calls the pairwise equivalence test EQUIV-P. Every call to EQUIV-P is conclusive and the result is stored either by merging the corresponding equivalence classes (lines 13–15), or updating `NEQ` (lines 16–18). Thus, each recursive call to EQUIV-P will avoid one iteration on the main loop of MIN-INCR by skipping (lines 7–10) that pair of states.

Finally, at lines 19–22, the set partition of the corresponding equivalence classes is created. Next, the DFA D is copied to D' and the equivalent states are merged by the call to `joinStates`. The last instruction, at line 25, returns the minimal DFA D', equivalent to D.

```
1  def EQUIV-P(p, q):
2      if (p, q) ∈ NEQ:
3          return False
4      if SET-SEARCH((p, q), PATH) ≠ nil:
5          return True
6      SET-INSERT((p, q), PATH)
7      for a ∈ Σ:
8          (p', q') = NORMALISE(FIND(δ(p, a)), FIND(δ(q, a)))
9          if p' ≠ q' and SET-SEARCH((p', q'), EQUIV) = nil:
10             SET-INSERT((p', q'), EQUIV)
11             if not EQUIV-P(p', q'):
12                 return False
13             else:
14                 SET-REMOVE((p', q'), PATH)
15     SET-INSERT((p, q), EQUIV)
16     return True
```

Listing 1.3. Pairwise equivalence test for MIN-INCR

Algorithm EQUIV-P, presented in Listing 1.3, is used to test the equivalence of the two states, p and q, passed as arguments.

The global variables `EQUIV` and `PATH` are updated with the pair (p, q) during each nested recursive call. As there is no recursion limit, EQUIV-P will only return when $p \nsim q$ (line 3) or when a cycle is found (line 5). If a call to EQUIV-P returns `False`, then all pairs of states recursively tested are distinguishable and variable `PATH` — used to store the sequence of calls to the transition function — will contain a set of distinguishable pairs of states. If it returns `True`, no pair of distinguishable states was found within the cycle and variable `EQUIV` will contain a set of equivalent states. This is the strategy which assures that each pair of states is tested for equivalence exactly once: every call to EQUIV-P is conclusive and the result stored for future use. It does, however, lead to an increased usage of memory.

The variables `EQUIV` and `PATH` are heavily used in EQUIV-P as several insert, remove, and membership-test operations are executed throughout the algorithm. In order to achieve the desired quadratic upper bound, all these operations must be performed in $O(1)$. Thus, we present in Listing 1.4 some efficient set representation and manipulation procedures.

```
1  def SET-MAKE(size):
2      HashTable = HASH-TABLE(size)
3      List = LIST()
4      return (HashTable, List)
5
6  def SET-INSERT(v, Set):
7      p0 = Set.HashTable[v]
8      LIST-REMOVE(p0, Set.List)
9      p1 = LIST-INSERT(v, Set.List)
10     Set.HashTable[v] = p1
11
12 def SET-REMOVE(v, Set):
13     p0 = Set.HashTable[v]
14     LIST-REMOVE(p0, Set.List)
15     Set.HashTable[v] = nil
16
17 def SET-SEARCH(v, Set):
18     if Set.HashTable[v] ≠ nil:
19         p = Set.HashTable[v]
20         return LIST-ELEMENT(p, Set.List)
21     else:
22         return nil
23
24 def SET-ELEMENTS(Set):
25     return Set.List
```

Listing 1.4. Set representation procedures

The set-manipulation procedures in Listing 1.4 simply combine a hash-table with a doubly-linked list. This is another space-time trade-off that allows us to assure the desired complexity on all operations. The hash-table maps a given value (state of the DFA) to the address on which it is stored in the linked list. Since we know the size of the hash-table in advance (n^2) searching, inserting, and removing elements is $O(1)$. The linked list assures that, at lines 14–15 and 17–18

of MIN-INCR, the loop is repeated only on the elements that were actually used in the calls to EQUIV-P, instead of iterating through the entire hash-table.

Theorem 1. *Algorithm MIN-INCR, in Listing 1.2, is terminating.*

Proof. It should suffice to notice the following facts:

- all the loops in MIN-INCR are finite;
- the variable `PATH` on EQUIV-P assures that the number of recursive calls is finite.

Lemma 1. *Algorithm EQUIV-P, in Listing 1.3, runs in $O(kn^2)$ time.*

Proof. The number of recursive calls to EQUIV-P is controlled by the local variable `PATH`. This variable keeps the history of calls to the transition function (line 8 in Listing 1.3). In the worst case, all possible pairs of states are used: $\frac{n^2}{2} - n$, due to the normalisation step. Since each call may reach line 7, we need to consider k additional recursive calls for each pair of states, hence $O(kn^2)$.

Lemma 2. *Algorithm EQUIV-P returns* `True` *if and only if the two states passed as arguments are equivalent.*

Proof. Algorithm EQUIV-P returns `False` only when the two states, p and q, used as arguments are such that $(p, q) \in$ `NEQ` (lines 2–3). This is correct because the global variable `NEQ` contains all the pairs of states already proven to be distinguishable. Conversely, EQUIV-P returns `True` only if $(p, q) \in PATH$ (lines 4–5) or a recursive call returned `True` (line 16). In both cases this means that a cycle with no distinguishable elements was detected, which implies that all the recursively visited pairs of states are equivalent.

Theorem 2. *Given a DFA $D = (Q, \Sigma, \delta, q, F)$, algorithm MIN-INCR computes the minimal DFA D' such that $D \sim D'$.*

Proof. Algorithm MIN-INCR finds pairs of equivalent states by exhaustive enumeration. The loop in lines 5–18 enumerates all possible pairs of states, and, for those not yet proven to be either distinguishable or equivalent, EQUIV-P is called. When line 19 is reached, all pairs of states have been enumerated and the equivalent ones have been found (cf. Lemma 2). The loop in lines 20–22 creates the equivalence classes and the procedure `joinStates`, at line 24, merges the equivalent states, updating the corresponding transitions. Since the new DFA D' does not have any equivalent states, it is minimal.

Lemma 3. *At the top-level call at line 13 in MIN-INCR, when EQUIV-P returns* `True`*, all the pairs of states stored in the global variable* `EQUIV` *are equivalent.*

Proof. By Lemma 2, if EQUIV-P returns `True` then the two states, p and q, used as arguments are equivalent. Since there is no depth recursion control, EQUIV-P only returns `True` when a cycle is detected. Thus being, all the pairs of states used as arguments in the recursive calls must also be equivalent. These pairs of states are stored in the global variable `EQUIV` at line 10 of EQUIV-P.

Lemma 4. *At the top-level call at line 13 in MIN-INCR, if EQUIV-P returns* `False`*, all the pairs of states stored in the global variable* `PATH` *are distinguishable.*

Proof. Given a pair of distinguishable states (p, q), clearly all pairs of states (p', q') such that $\delta(p', w) = p$ and $\delta(q', w) = q$ are also distinguishable, for $w \in \Sigma^\star$. By Lemma 2, EQUIV-P returns `False` only when the two states, p and q, used as arguments are distinguishable. Throughout the successive recursive calls to EQUIV-P, the global variable `PATH` is used to store the history of calls to the transition function (line 6) and thus contains only pairs of states with a path to (p, q). All of these pairs of states are therefore distinguishable.

Lemma 5. *Each time that EQUIV-P calls itself recursively, the two states used as arguments will not be considered in the main loop of MIN-INCR.*

Proof. The arguments of every call of EQUIV-P are kept in two global variables: `EQUIV` and `PATH`.

By Lemma 3, whenever EQUIV-P returns `True`, all the pairs of states stored in `EQUIV` are equivalent. Immediately after being called from MIN-INCR (line 13), if EQUIV-P returns `True`, the equivalence classes of all the pairs of states in `EQUIV` are merged (lines 14–15). Future references to any of these pairs will be skipped at lines 9–10.

In the same way, by Lemma 4, if EQUIV-P returns `False`, all the pairs of states stored in `PATH` are distinguishable. Lines 17–18 of MIN-INCR update the global variable `NEQ` with this new information and future references to any of these pairs of states will be skipped at lines 7–8 of MIN-INCR.

Theorem 3. *Algorithm MIN-INCR is incremental.*

Proof. Halting the main loop of MIN-INCR at any point within the lines 5–18 only prevents the finding of *all* the equivalent pairs of states. Merging the known equivalent states on D', a copy of the input DFA D, assures that the size of D' is not greater than that of D and thus, is closer to the minimal equivalent DFA. Calling MIN-INCR with D' as the argument would resume the minimisation process, finding the remaining equivalent states.

Theorem 4 (Main result). *Algorithm MIN-INCR, in Listing 1.2, runs in* $O(kn^2\alpha(n))$ *time.*

Proof. The number of iterations of the main loop in lines 5–18 of MIN-INCR is bounded by $\frac{n^2}{2} - n$, due to the normalisation step. Each iteration may call EQUIV-P, which, by Lemma 1, is $O(kn^2)$. By Lemma 5 every recursive call to EQUIV-P avoids one iteration on the main loop. Therefore, disregarding the UNION-FIND operations and because all operations on variables `NEQ`, `EQUIV`, and `PATH` are $O(1)$, the $O(kn^2)$ bound holds. Since there are $O(kn^2)$ FIND and UNION intermixed calls, and exactly n MAKE calls, the time spent on all the UNION-FIND operations is bounded by $O(kn^2\alpha(n))$ — cf. Subsection 2.1. All things considered, MIN-INCR runs in $O(kn^2 + kn^2\alpha(n)) = O(kn^2\alpha(n))$.

Corollary 1. *Algorithm MIN-INCR runs in* $O(kn^2)$ *time for all* practical *values of* n.

Proof. Function α is related to an inverse of Ackermann's function. It grows so slowly ($\alpha(16^{512}) \leq 4$) that we may consider it a constant.

5 Experimental Results

In this Section we present some comparative experimental results on four DFA minimisation algorithms: Brzozowski, Hopcroft, Watson, and the new proposed incremental one. The results are presented on Table 1, Table 2, and Table 3.

All algorithms are implemented in the `Python` programming language and integrated in the **FAdo** project [FAd10]. The tests were executed in the same computer, an Intel® Xeon® 5140 at 2.33 GHz with 4 GB of RAM, running a minimal 64 bit Linux system. We used samples of 20.000 automata, with $n \in \{5, 10, 50, 100\}$ states and alphabets with $k \in \{2, 10, 25, 50\}$ symbols. Since the data sets were obtained with a uniform random generator [AMR07, AAA+09], the size of each sample is more than enough to ensure results statistically significant with a 99% confidence level within a 1% error margin. The sample size is calculated with the formula $N = (\frac{z}{2\epsilon})^2$, where z is obtained from the normal distribution table such that $P(-z < Z < z)) = \gamma$, ϵ is the error margin, and γ is the desired confidence level.

Table 1. Experimental results for ICDFAs with $n \in \{5, 10\}$ states

$n = 5$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	1424.50	4	119.71	4	45.24	4	22.66	4
Hopcroft	3442.34	4	980.39	4	469.37	4	238.46	4
Watson	3616.63	4	573.88	4	54.29	4	8.00	4
Incremental	4338.39	4	2762.43	4	1814.88	4	1091.70	4
$n = 10$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	73.45	4	1.89	4	0.50	3248	0.21	1912
Hopcroft	1757.46	4	250.46	4	100.95	4	49.74	4
Watson	691.80	4	0.01	4	0.00	4	0.00	4
Incremental	2358.49	4	1484.78	4	885.73	4	488.75	4

Each test was given a time slot of 24 hours. Processes that did not finish within this time limit were killed. Thus, and because we know how many ICDFAs were in fact minimised before each process was killed, the performance of the algorithms is measured in *minimised ICDFAs per second* (column **Perf.**). We also include a column for the memory usage (**Space**), which measures the peak value (worst-case) for the minimisation of the 20.000 ICDFAs in *kilobytes*.

Clearly, the new incremental method always performs better. Both Brzozowski and Watson's algorithm clearly show their exponential character, being in fact, the only two algorithms that did not finish several minimisation tests. Hopcroft's

Table 2. Experimental results for ICDFAs with $n \in \{50, 100\}$ states

$n = 50$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	0.00	664704	0.00	799992	0.00	1456160	0.00	750312
Hopcroft	39.48	4	6.31	4	2.45	4	1.21	4
Watson	0.00	4	0.00	4	0.00	4	0.00	4
Incremental	117.21	288	94.33	540	73.94	612	53.31	636
$n = 100$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	0.00	2276980	0.00	1061556	0.00	961144	0.00	2862312
Hopcroft	6.25	4	0.97	4	0.37	4	0.18	4
Watson	0.00	4	0.00	4	0.00	4	0.00	4
Incremental	28.23	1028	24.94	2452	21.58	3444	17.17	3824

Table 3. Experimental results for ICDFAs with 1000 states

$n = 1000$						
	$k = 2$		$k = 3$		$k = 5$	
	Perf.	Space	Perf.	Space	Perf.	Space
Hopcroft	0.0121	4	0.0074	4	–	–
Incremental	0.2616	34296	0.2416	34068	0.2411	33968

algorithm, although presenting a behaviour that appears to be very close to its worst-case ($O(kn \log(n))$, is always slower than the quadratic incremental method.

Because of the big difference between the measured performance of Hopcroft's algorithm and the new quadratic incremental one, we ran a new set of tests, only for these algorithms, using the largest ICDFA samples we have available. The results are presented on Table 3. These tests were performed on the exact same conditions as the previously described ones but each process was allowed to execute for 96 hours. Surprisingly, while Hopcroft's algorithm did not finish any of the batches within this time limit, it took only a little over 23 hours for the quadratic algorithm to minimise the sample of 20.000 ICDFAs with 1000 states and 5 symbols. The memory usage of the incremental algorithm, however, is far superior. It always required nearly 33 MB while Hopcroft's algorithm did not use more than 4 kB.

6 Conclusions

We presented a new incremental minimisation algorithm. Unlike other non-incremental minimisation algorithms, the intermediate results are usable and reduce the size of the input DFA. This property can be used to minimise a DFA when it is simultaneously processing a string or, for example, to reduce the size of a DFA when the running-time of the minimisation process must be restricted for some reason.

We believe that this new approach, while presenting a quadratic worst-case running-time, is quite simple and easy to understand and to implement. According to the experimental results, this minimisation algorithm outperforms

Hopcroft's $O(kn\log(n))$ approach, at least in the average case, for reasonably sized automata.

References

[AAA+09] Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: FAdo and GUItar: tools for automata manipulation and visualization. In: Maneth, S. (ed.) CIAA 2009. LNCS, vol. 5642, pp. 65–74. Springer, Heidelberg (2009)

[AMR07] Almeida, M., Moreira, N., Reis, R.: Enumeration and generation with a string automata representation. Theoret. Comput. Sci. 387(2), 93–102 (2007); Special issue Selected papers of DCFS 2006

[Brz63] Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Fox, J. (ed.) Proc. of the Sym. on Math. Theory of Automata, NY. MRI Symposia Series, vol. 12, pp. 529–561 (1963)

[CLRS03] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT, Cambridge (2003)

[FAd10] Project FAdo. FAdo: tools for formal languages manipulation, http://www.ncc.up.pt/FAdo (access date:1.1.2010)

[HMU00] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison Wesley, Reading (2000)

[Hop71] Hopcroft, J.: An $n\log n$ algorithm for minimizing states in a finite automaton. In: Proc. Inter. Symp. on the Theory of Machines and Computations, Haifa, Israel, pp. 189–196. Academic Press, London (1971)

[Huf55] Huffman, D.A.: The synthesis of sequential switching circuits. The Journal of Symbolic Logic 20(1), 69–70 (1955)

[Moo58] Moore, E.F.: Gedanken-experiments on sequential machines. The Journal of Symbolic Logic 23(1), 60 (1958)

[Tar75] Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. JACM 22(2), 215–225 (1975)

[Wat95] Watson, B.W.: Taxonomies and toolkit of regular languages algortihms. PhD thesis, Eindhoven Univ. of Tec. (1995)

[Wat01] Watson, B.W.: An incremental DFA minimization algorithm. In: International Workshop on Finite-State Methods in Natural Language Processing, Helsinki, Finland (August 2001)

[WD03] Watson, B.W., Daciuk, J.: An efficient DFA minimization algorithm. Natural Language Engineering, 49–64 (2003)

Finite Automata for Generalized Approach to Backward Pattern Matching*

Jan Antoš[1] and Bořivoj Melichar[2]

[1] Department of Computer Science & Engineering, Faculty of Electrical Engineering, Czech Technical University, Karlovo nám. 13, 121 35 Prague 2, Czech Republic
[2] Department of Theoretical Science, Faculty of Information Technology, Czech Technical University, Kolejní 2, 160 00 Prague 6, Czech Republic
antosj@fel.cvut.cz, melichar@fit.cvut.cz

Abstract. We generalized the DAWG backward pattern matching approach to be able to solve a broad range of pattern matching problems. We use a definition of a class of problems. We describe a finite automaton for the basic pattern matching problem of finding an exact occurrence of one string in a text. We propose a mechanism to use simple operations over finite automata in a systematic approach to derive automata for solving problems from a defined class, such as approximate matching, regular expression matching, sequence matching, matching of set of patterns, etc. and their combinations. The benefit of this approach is the ability to quickly derive solutions for newly formulated pattern matching problems.

Keywords: backward pattern matching, finite automata theory, automata construction, approximate pattern matching, classification.

1 Introduction

1.1 Historical Context

The backward pattern matching is a discipline of pattern matching using a mechanism invented by Boyer and Moore in 1977 [4]. It speeds up the text processing by skipping parts of a text. Its lower bound of time complexity is $O(n/m)$ where n is the length of the text and m is the length of a pattern. Many algorithms for backward pattern matching appeared in later years [8]. One of these is Backward DAWG Matching (BDM) [12] that uses a finite automaton called Directed Acyclic Word Graph (DAWG).

The pattern matching includes not only a matching of a single string in a text but also a broad range of other problems as a matching of a finite [1] or infinite [11] set of patterns, approximate matching based on Hamming [9] and Levenshtein [13] distances, sequence matching [5], don't care symbols and regular expressions [11], etc. In [15] a 6D classification was described classifying 192 pattern matching problems into one class.

* This research has been partially supported by the research program MSMT 6840770014 and by The Czech Science Foundation as project No. 201/09/0807.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 49–58, 2011.

1.2 Motivation

The motivation of this paper is to describe an approach for quick creation of new solutions to newly formulated pattern matching problems. If we start with the BDM-like algorithm to solve the exact pattern matching of one string in a text, then our motivation is to define algorithm(s) to derive similar solutions for the mentioned broad range of problems from this basic algorithm.

The motivation is not to present a performance improvement over existing algorithms for specific pattern matching problems, but rather to find a process to derive new algorithms for new problems.

2 Basic Definitions

Definition 1 (Alphabet). An alphabet *A is a finite non-empty set of symbols. The number of symbols will be denoted by $|A|$.*

Definition 2 (String). *A* string *over the given alphabet A is a finite sequence of symbols of the alphabet A. The* length of a string *w is the number of symbols in the string w and is denoted by $|w|$. A* substring *of string s, where the substring starts at position i of the string s and ends at position j (inclusive), will be denoted as $s_i \dots s_j$. For a string $s = s_1 \dots s_n$ where $n = |s|$ a* reversed string *is string $s^R = s_n \dots s_1$.*

Definition 3 (Factor, prefix, suffix, antifactor). *A string x is said to be a* factor (substring) *of string y if $y = uxv$ for some strings u, v, a* prefix *of y if $u = \varepsilon$, a* suffix *of y if $v = \varepsilon$ and* antifactor *if x is not a factor of y. The set of all factors, prefixes, suffixes and antifactors of a string s will be denoted by $fact(s)$, $pref(s)$, $suff(s)$ and $antifact(s)$ respectively.*

Definition 4 (Proper prefix, proper suffix). A proper prefix *(*proper suffix*) of a string is a prefix (suffix) that is not equal to the string itself and not empty. The set of all proper prefixes and proper suffixes of string s will be denoted by $pref^+(s)$ and $suff^+(s)$ respectively. Functions $fact$, $pref$, $suff$, $pref^+$ and $suff^+$ are also defined for the set of strings S as $fact(S) = \bigcup_{s \in S} fact(s)$.*

Definition 5 (Projection of Pattern, Image of Pattern). The projection of a pattern *is a function that, for a given pattern matching problem θ and a given pattern p outputs a set of all strings in an alphabet A that are considered a match to the given pattern. We will denote this function $proj_{\theta,A}(p)$ or simply $proj(p)$. For a set of patterns P the function is defined as $proj(P) = \bigcup_{p \in P} proj(p)$. Strings produced by a projection are called* pattern images*.*

Remark 1. Examples of pattern matching problems are: exact matching of one string, of a finite set of strings, approximate matching, matching of all substrings, etc. *An instance of a pattern matching problem* is a combination of a problem and a set of patterns. For example: an instance of the problem "exact matching of one string" and the pattern "banana" is "exact matching of string banana".

Remark 2. A pattern is denoted by p, set of patterns by P, a pattern image by g and set of pattern images by G.

3 Background

3.1 Backward Pattern Matching

The principle of backward pattern matching is to examine whether a pattern occurs in a sliding window in a text. By comparing symbols from right to left only a part of the window is checked to decide. The matching time is sub-linear with the length of text. Several approaches exist. Figure 1 (adapted from [16]) shows *good-prefix shift* approach [7] used in this paper.

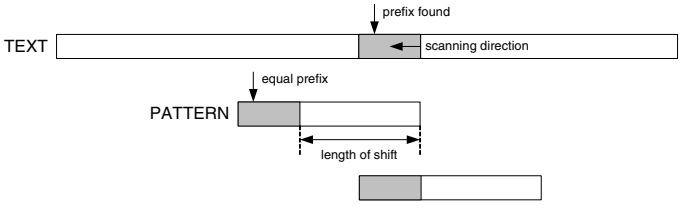


Fig. 1. The principle of backward pattern matching using the good-prefix shift method

3.2 Classification of Pattern Matching Problems

To specify a pattern matching problem we use the six-dimensional classification of pattern matching problems presented in [15]. A problem is specified as a point in 6D space (Figure 2). The points are referenced by six-letter abbreviations of dimension values. An example is SFFRCO which stands for [String matching, Full pattern, Finite number of patterns, approximate matching with Replace edit distance (i.e. Hamming distance), take Care of all symbols, One string].

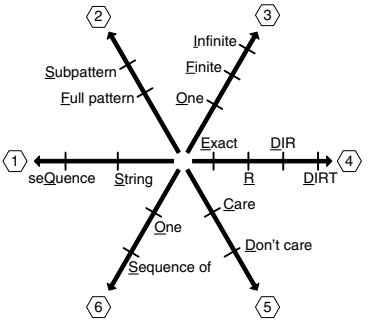


Fig. 2. The six-dimensional classification of pattern matching problems

The proposed approach solves problems from 6D space as approximate pattern matching, matching of finite or infinite set of patterns, don't care symbols, sub patterns, searching for sequences, etc. and their combinations (as approximate

matching of infinite set of patterns given by a regular expression). For some of these 192 combinations a backward matching approach does not yet exist. New problems can be solved by extending the 6D space.

4 Backward Pattern Matching Machine

For full details please refer to [3]. The processing in BPMM is split into two parts (Figure 3). The *constructor* takes a problem and a set of patterns as inputs and creates an extended deterministic finite automaton representing an instance of pattern matching problem. The *executor* uses the automaton over a text to do the pattern matching.

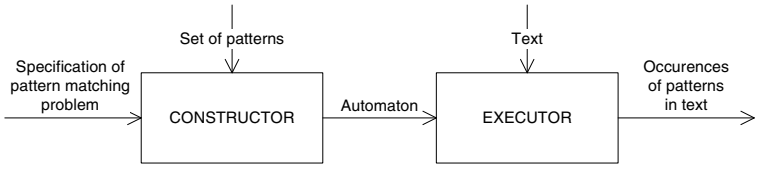


Fig. 3. The schema of the universal Backward Pattern Matching Machine

4.1 Implementation Options

BPMM can be constructed using various implementation options. The good-prefix shift method [7] is used to compute the shift function in this paper (other options are repeated-suffix shift method [7], antifactor shift [16], etc.) To decide whether to continue scanning the text in the sliding window a $f = fact(G)$ function is used (other options are $suff(G)$, $factoracle(G)$ [2] and others [6]).

To find prefixes of pattern images, a suffix automaton M, $L(M) = suff(G^R)$ is used because the text is read backwards: $(pref(G))^R = suff(G^R)$. The same automaton is used to implement f as well. It holds that $suff(G^R) \subset fact(G^R)$ and $antifact(G^R) \cap suff(G^R) = \emptyset$. The language accepted by M is not f but the transition function for $w \in f$ is defined and for $w \notin f$ is not: $w \in fact(G) \Rightarrow |\delta(q_0, w^R)| \geq 1$ while $w \notin fact(G) \Rightarrow \delta(q_0, w^R) = \emptyset$.

4.2 Executor

The executor's algorithm (Algorithm 1) is the same for all pattern matching problems from the studied class.

Variable *position* stores the index of the last symbol of a sliding window in the text. Text is indexed from 1. *Offset* is counted from *position* toward the beginning of the text. *Transition counter (tc)* tracks the number of transitions from the last initialization of the automaton. To process each text window the automaton is set to its initial state. It then reads the text backwards. A pattern is found if a final state $q \in F_p \cup F_{ps}$ is reached. A prefix of a pattern is found

Algorithm 1: The executor algorithm
Input: Text T, description of problem instance $M = (Q, A, \delta, q_0, F_p, F_s, F_{ps})$, maximum safe shift distance $shift_{max} = |G|_{min}$
Output: Positions of the first symbols of patterns $p \in P$ in text T
Method:

```
1   q ← q0, position ← shift_max, offset ← 0
2   plc ← 0        (note: plc = prefix length counter)
3   tc ← 0         (note: tc = transition counter)
4   while position ≤ |T| do
5       if δ(q, T_{position−offset}) ≠ ∅ then
6           q ← δ(q, T_{position−offset}), tc ← tc + 1
7           if q ∈ F_s ∪ F_ps then plc ← tc endif
8           if q ∈ F_p ∪ F_ps then output(position − offset) endif
9           offset ← offset + 1
10      else
11          shift ← max{1, shift_max − plc}
12          position ← position + shift
13          offset ← 0, plc ← 0, tc ← 0, q ← q0
14      endif
15  endwhile
```

if $q \in F_s \cup F_{ps}$ is reached. *Prefix length counter (plc)* keeps track of the longest prefix in the current window. When there is no transition defined for the input symbol an antifactor has been found and the window is shifted.

The formal proof of the algorithm's correctness is given in [3]. The memory and the time complexity is also studied in [3]. The memory complexity of this algorithm is constant with the length of text and the time complexity has the upper bound of $O(n|G|_{max})$ and the lower bound of $O(n/|G|_{min})$ where n is the length of the text and G is the set of pattern images. The average time complexity depends on the problem solved. Optimizations leading to the upper bound of $O(n)$ are proposed in [3] for specific subclasses of problems.

4.3 Constructor

The output of the constructor is a Backward Pattern Matching Automaton (BPMA) representing an instance of pattern matching problem. For the same set of patterns different problems produce different BPMAs.

Definition 6 (Backward Pattern Matching Automaton). The Backward Pattern Matching Automaton *is a seven-tuple* $M = (Q, A, \delta, q_0, F_p, F_s, F_{ps})$, *where* Q *is a finite set of states,* A *is a finite input alphabet,* δ *is a mapping* $Q \times A \to Q$, $q_0 \in Q$ *is the initial state,* $F_p \subset Q$, $F_s \subset Q$, $F_{ps} \subset Q$ *are mutually disjoint sets of final states, i.e.* $(F_p \cap F_s = \emptyset) \wedge (F_p \cap F_{ps} = \emptyset) \wedge (F_s \cap F_{ps} = \emptyset)$. *Sets* F_p, F_s *and* F_{ps} *are such, that if automaton* $M_p = (Q, A, \delta, q_0, F_p)$ *is accepting* $L(M_p)$ *then automaton* $M_s = (Q, A, \delta, q_0, F_s)$ *is accepting* $L(M_s) = suff^+(L(M_p)) \setminus L(M_p)$ *and automaton* $M_{ps} = (Q, A, \delta, q_0, F_{ps})$ *is accepting* $L(M_{ps}) = suff^+(L(M_p)) \cap L(M_s)$.

The BPMA has three disjoint sets of final states accepting three languages L_p, L_s and L_{ps} that together represent the language of all reversed pattern images G^R and of all suffixes of all reversed pattern images $suff^+(G^R)$:

$$L_p = G^R \setminus suff^+(G^R), L_s = suff^+(G^R) \setminus G^R, L_{ps} = G^R \cap suff^+(G^R).$$

The L_s language accepts pattern matches, L_p accepts prefixes of pattern matches and L_{ps} accepts prefixes that are at the same time pattern matches.

Remark 3. In transition diagrams, a state $q \in F_p$ is denoted by a full inner circle, $q \in F_s$ by a dotted inner circle and $q \in F_{ps}$ by a dashed inner circle.

4.4 Construction of a Backward Pattern Matching Automaton

The construction has two steps:

1. For a problem θ and a set of patterns P we construct a Reversed Projection Automaton (RPA). RPA is nondeterministic finite automaton M_{RPA} accepting a set of all reversed pattern images $L(M_{RPA}) = \big(proj(P)\big)^R$.
2. From M_{RPA} we construct a BPMA by a problem-independent algorithm.

4.5 Construction of a Reversed Projection Automaton

A RPA for a specific problem is not constructed by a problem-specific algorithm. The 6D classification is used to construct the automaton one step at time:

1. Construct automaton for Dimension 3: One - Finite - Infinite
2. Modify the automaton from step 1 for Dimension 5: Care - Don't care
3. Modify the automaton from step 2 for Dimension 4: Exact - R - DIR - DIRT
4. Modify the automaton from step 3 for Dimension 2: Full pattern - Subpattern
5. Modify the automaton from step 4 for Dimension 1: seQuence - String
6. Modify the automaton from step 5 for Dimension 6: One - Sequence of

At first an automaton for a point on the 3^{rd} axis (matching of one, finite or infinite number of patterns) is constructed by one of three algorithms. The result is RPA for one of SFOECO, SFFECO or SFIECO problems.

Next step is the 5^{th} axis: If one string with "don't care" symbols is to be matched, a SFOECO automaton is converted to SFOEDO automaton. In the same way, SFFECO automaton is converted to SFFEDO and SFIECO to SFIEDO. The same algorithm is used for all three cases. By using at most two algorithms (out of four: three for the 3^{rd} axis and one for the 5^{th} axis) a RPA for six problems can be constructed. The 4^{th} axis is processed next and so on through the 6D space (or any other defined problem space).

At most six algorithms (out of ten) are needed to construct RPAs for all 192 problems. The same method and different classification will build RPAs for different problems.

The approach is demonstrated here by giving algorithms needed to construct BPMA for SFIERCO problem (approximate pattern matching of a regular

expression). More algorithms are given in [3]. The algorithms are very straightforward, yet their combination can solve complex problems.

4.6 Example: Construction of RPA for SFIERCO Problem

Let us have the regular expression $v = (aaaa)^+b$. We first construct RPA for Dim 3 (infinite set of patterns). The RPA will accept language $L(RPA) = v^R$. We construct it by one of well known algorithms [10] (Figure 4).

Then we modify it by Algorithm 2 for Dim 4 (approximate matching by Hamming distance). The result is RPA for SFIERCO problem (Figure 5).

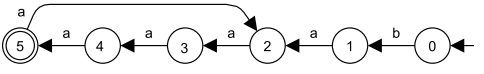


Fig. 4. Reversed projection automaton for regular expression $v = (aaaa)^+b$

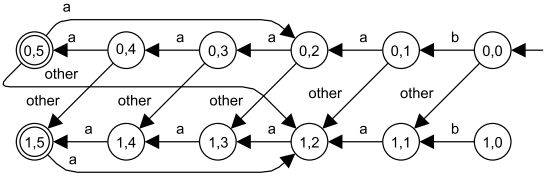


Fig. 5. RPA for regular expression $v = (aaaa)^+b$ and Hamming distance with $k_{max} = 1$

Algorithm 2: Construction of automaton for Dimension 4 for R-matching
Input: Automaton $M_{input} = (Q, A, \delta, q_0, F)$, Hamming distance k_{max}
Output: Nondeterministic finite automaton M
Method:

1 $Q' \leftarrow \emptyset,\ F' \leftarrow \emptyset$
2 **for** $\forall k \in \langle 0, k_{max} \rangle$ **do**
3 $\quad Q' \leftarrow Q' \cup \{q_{k,i} : q_i \in Q\}$
4 $\quad \delta'(q_{k,i}, a) \leftarrow \delta'(q_{k,i}, a) \cup \{q_{k,j} : q_j \in \delta(q_i, a)\}$ for all $a \in A,\ q_i \in Q$
5 $\quad F' \leftarrow F' \cup \{q_{k,i} : q_i \in F\}$
6 **endfor**
7 **for** $\forall k \in \langle 0, k_{max} - 1 \rangle$ **do**
8 $\quad \delta'(q_{k,i}, \overline{a}) \leftarrow \delta'(q_{k,i}, \overline{a}) \cup \{q_{k+1,j} : q_j \in \delta(q_i, a)\}$ for all $a \in A,\ q_i \in Q$
9 **endfor**
10 $M \leftarrow (Q', A, \delta', q_{0,0}, F')$

4.7 Construction of Backward Pattern Matching Automaton

A BPMA is constructed from RPA by Algorithm 3. We assume the RPA has initial state with no incoming transitions.

Algorithm 3: Construction of nondeterministic BPMA
Input: Nondeterministic finite automaton $M_{RPA} = (Q, A, \delta, q_0, F)$, $Q = \{q_0, q_1, \cdots, q_n\}$, q_0 has no incoming transitions
Output: Extended nondeterministic finite automaton M
Method:

```
1   Q' ← {q_{i,p}, q_{i,s} : i ∈<0, n>}
2   for ∀i, j ∈<0, n>, ∀a ∈ A do
3       if δ(q_i, a) ∋ q_j then
4           if i ≠ 0 then
5               δ'(q_{i,p}, a) ← δ'(q_{i,p}, a) ∪ q_{j,p}
6               δ'(q_{i,s}, a) ← δ'(q_{i,s}, a) ∪ q_{j,s}
7           else
8               δ'(q_{0,p}, a) ← δ'(q_{0,p}, a) ∪ q_{j,p}
9               δ'(q_{0,p}, a) ← δ'(q_{0,p}, a) ∪ ⋃_{q_k ∈ Q\{q_0}} δ(q_k, a)
10          endif
11      endif
12  endfor
13  F_p ← {q_{i,p} : q_i ∈ F, i ∈<0, n>}, F_s ← {q_{i,s} : q_i ∈ F, i ∈<0, n>}, F_ps ← ∅
14  M ← (Q', A, δ', q_{0,p}, F_p, F_s, F_ps)
```

The algorithm creates a new automaton with two sets of states - states with the second index p accept reversed patterns, states with the second index s accept proper suffixes of reversed patterns. The result looks like the union of a pattern automaton and a suffix automaton. The suffix automaton is created by virtually adding the ε-transitions from the initial state to all s-indexed states. (All ε-transitions are replaced by the equivalent non-epsilon transitions on line 9.)

The output of Algorithm 3 is nondeterministic automaton but in the executor we use a deterministic finite automaton. Well known algorithms for determination and minimization of finite automata (see [10]) can be used. They only need to be adapted to work with the tree disjoint set of states, see [3].

4.8 Time and Space Complexity

The construction algorithms for RPA given in [3] run in linear time with the length (or sum of lengths) of patterns. Also the size of the automaton is linear. The complexity for SFIECO problem is given by the conversion of a regular expression to an automaton [10].

The construction of BPMA (Alg. 3) has time complexity of $O(|A|x^2)$ where x is the number of states of the input automaton: the for cycle (line 2) can be implemented by a tree-traversal algorithm [10] to run at most x times. Line 9 can be implemented to iterate over incoming transitions (at most $x|A|$ times).

It is well known that the conversion from nondeterministic (NFA) to deterministic (DFA) automaton requires at most $O(2^x)$ time where x is the number of states in NFA. The resulting DFA may have at most $O(2^x)$ states. For some problems it has been shown that the upper bound is lower [14] but in general it still needs to be studied. Yet for typical cases the size of BPMA is practical.

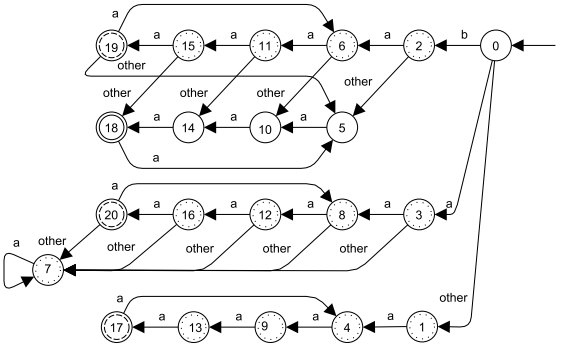


Fig. 6. BPMA (after minimization and determinization) for regular expression $v = (aaaa)^+b$ and Hamming distance with $k = 1$

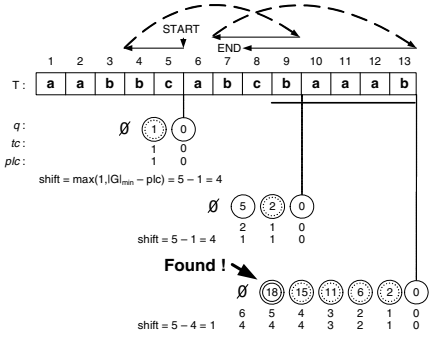


Fig. 7. Example run of the executor: approximate matching of regular expression $v = (aaaa)^+b$ in text

5 Conclusion

The proposed approach is a generalization of backward pattern matching. The one-step-at-a-time approach for construction of Reversed Projection Automaton (RPA) has the benefit of easy extension to new problems and their combination with currently known problems. The BPMM Executor's pattern matching algorithm is universal to solve any problem described by RPA. The benefit is the ability to quickly derive pattern matching solutions for new problems.

The pattern matching algorithm is not optimal but it can be optimized for specific problem subclasses [3]. Future work will study in more detail the size of BPMA (due to the determinization of NFA) to precisely characterize under which conditions the approach might not be practical due to the size of the automaton.

References

1. Aho, A.V., Corasick, M.J.: Efficient String Matching: An Aid to Bibliographic Research. Communications of ACM 18(6), 333–340 (1975)
2. Allauzen, C., Crochemore, M., Raffinot, M.: Factor Oracle: A New Structure for Pattern Matching. In: Bartosek, M., Tel, G., Pavelka, J. (eds.) SOFSEM 1999. LNCS, vol. 1725, pp. 295–306. Springer, Heidelberg (1999)
3. Antoš, J.: Automaton-based Backward Pattern Matching. Dissertation thesis. CTU in Prague (2010), http://www.stringology.org/papers/Antos-PhD_thesis-2010.pdf
4. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. C. ACM 20(10), 762–772 (1977)
5. Chvatal, V., Klarner, D.A., Knuth, D.E.: Selected Combinatorial Research Problems. STAN-CS-72-292, Stanford University (1972)
6. Cleophas, L., Watson, B.W., Zwaan, G.: Automaton-based sublinear keyword pattern matching. In: Proceedings of the 11th SPIRE, Padova, Italy (2004)
7. Crochemore, M., Czumaj, A.: Gąsieniec, L., et al.: Deux méthodes pour accélerer l'algorithme de Boyer-Moore. In: Actes des 2e Journées franco-belges: Théories des Automates et Applications, pp. 45–63. Public. de l'Univ. de Rouen, No. 176 (1991)
8. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)
9. Hamming, R.W.: Error-detecting and error-correcting codes. Bell System Technical Journal 29(2), 147–160 (1950)
10. Hopcroft, J.E., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, Reading (2001)
11. Kleene, S.C.: Representation of Events in Nerve Nets and Finite Automata. In: Automata Studies, pp. 3–42. Princeton University Press, Princeton (1956)
12. Lecroq, T.: A variation on the Boyer-Moore algorithm. Theoretical Computer Science 92(1), 119–144 (1992)
13. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Doklady Akademii Nauk SSSR 163(4), 845–848 (1965)
14. Melichar, B.: String Matching with k Differences by Finite Automata. In: Proceedings of the 13th ICPR, vol. II, pp. 256–260 (1996)
15. Melichar, B., Holub, J.: 6D Classification of Pattern Matching Problems. In: Proceedings of PSC 1997, Prague, Czech republic, pp. 24–32 (1997)
16. Melichar, B., Holub, J., Polcar, T.: Text Searching Algorithms, vol. I, II (2005), http://psc.felk.cvut.cz/athens/

Partial Derivative Automata Formalized in Coq

José Bacelar Almeida[3], Nelma Moreira[1],
David Pereira[1], and Simão Melo de Sousa[2]

[1] DCC-FC & LIACC, University of Porto
Rua do Campo Alegre 1021, 4169-007, Porto, Portugal
{nam,dpereira}@ncc.up.pt
[2] LIACC & DI, University of Beira Interior
Rua Marquês d'Ávila e Bolama, 6201-001 Covilhã, Portugal
desousa@di.ubi.pt
[3] CCTC & DI, University of Minho,
Campus de Gualtar, 4710-057 Braga, Portugal
jba@di.uminho.pt

Abstract. In this paper we present a computer assisted proof of the correctness of a partial derivative automata construction from a regular expression within the Coq proof assistant. This proof is part of a formalization of Kleene algebra and regular languages in Coq towards their usage in program certification.

1 Introduction

The use of proof assistants has gained increasing importance in mathematics and computer science. Their value in the assurance of theorem and algorithm correctness is obvious, since all the steps and intricacies involved in the proof process are formally and mechanically checked. The use of the Coq proof assistant for program verification is specially attractive because correctness proofs can be compiled as *proof certificates*, and the constructive components of the specification and proof development can be extracted into functional programs.

In this paper we describe a formalization of regular languages in Coq. Our main result is the proof of the correctness of a partial derivative automata construction from a regular expression. This result is a step towards the implementation of a proved terminating, and correct, decision procedure for regular expression equivalence based on the notion of (partial) derivatives. From such implementation it is possible to extract a *correct-by-construction* functional program, and it is also possible to develop *proof tactics* that automate the construction of proofs. Kleene algebra can be used to capture several properties of programs. In this setting, testing Kleene algebra terms equivalence can correspond to proving partial correctness of programs. Defining and proving the correctness of a decision procedure within a proof assistant that features *proof objects*[1] allows to obtain certificates that facilitate the automation of formal software verification.

[1] In such systems proof objects are values that can be compiled into binary code.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 59–68, 2011.

The paper is organised as follows. In Section 2 we review some definitions about regular languages and finite automata. The partial derivative automaton and Mirkin's construction are reviewed in Section 3. Section 4 presents a small introduction to the Coq proof assistant. In Section 5 we describe the formalization of regular languages in Coq and present the main result. In Section 6 we comment on related work. Finally, in Section 7 we draw some conclusions and point some future work.

2 Regular Languages and Finite Automata

Let $\Sigma = \{a_1, a_2, \ldots, a_k\}$ be an *alphabet* (set of *symbols*). A *word* w over Σ is any finite sequence of symbols. The *empty word* is denoted by ε. The *concatenation* of two words w_1 and w_2 is the word $w = w_1 w_2$. Let $\Sigma^\star$ be the set of all words over Σ. A *language* over Σ is a subset of $\Sigma^\star$. If L_1 and L_2 are two languages, then $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$. The operator $\cdot$ is often omitted. The *power of a language* L is inductively defined by $L^0 = \{\varepsilon\}$, $L^n = LL^{n-1}$, for $n \geq 1$. The Kleene *star* $L^\star$ of a language L is $\cup_{n \geq 0} L^n$. Given a word $w \in \Sigma^\star$ and $L \in \Sigma^\star$, the (*left-*)*quotient* of L by w is $w^{-1}L = \{v \mid wv \in L\}$.

A *regular expression* (re) α over Σ represents a *regular language* $\mathcal{L}(\alpha) \subseteq \Sigma^\star$ and is inductively defined by: $\emptyset$ is a re and $\mathcal{L}(\emptyset) = \emptyset$; ε is a re and $\mathcal{L}(\varepsilon) = \{\varepsilon\}$; $a \in \Sigma$ is a re and $\mathcal{L}(a) = \{a\}$; if α_1 and α_2 are re, $(\alpha_1 + \alpha_2)$, $(\alpha_1\alpha_2)$ and $(\alpha_1)^\star$ are re, respectively with $\mathcal{L}((\alpha_1 + \alpha_2)) = \mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2)$, $\mathcal{L}((\alpha_1\alpha_2)) = \mathcal{L}(\alpha_1)\mathcal{L}(\alpha_2)$ and $\mathcal{L}((\alpha_1)^\star) = \mathcal{L}(\alpha_1)^\star$. The *alphabetic size* of a re α is the number of symbols of α and it is denoted by $|\alpha|_\Sigma$. The *constant part* of a re is denoted by $\varepsilon(\alpha)$ and defined by $\varepsilon(\alpha) = \varepsilon$ if $\varepsilon \in \mathcal{L}(\alpha)$, and $\varepsilon(\alpha) = \emptyset$ otherwise. The same function can be applied to languages. Let RE be the set of regular expressions over Σ. If two re's α and β are syntactically identical, we write $\alpha \equiv \beta$. Two re's are equivalent if they represent the same regular language, that is, if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$, and we write $\alpha = \beta$. The equational properties of regular expressions are axiomatically captured by a *Kleene algebra* (KA), normally called *the algebra of regular events*, after the seminal work of S. C. Kleene [Kle56]. A KA is an algebraic structure $(K, 0, 1, +, \cdot, ^\star)$ such that $(K, 0, 1, +, \cdot)$ is an *idempotent semiring* and where the operator * (Kleene's *star*) is characterized by a set of axioms. The algebra of regular events is given by $(\mathsf{RE}, \emptyset, \varepsilon, +, \cdot, ^\star)$. There are several ways of axiomatizing a KA but we considered the one presented by D. Kozen in [Koz94].

A *non-deterministic finite automaton* (NFA) $\mathcal{A}$ is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, q_0 the initial state and $F \subseteq Q$ is the set of final states. For $q \in Q$ and $a \in \Sigma$, we denote the set $\{p \mid (q, a, p) \in \delta\}$ by $\delta(q, a)$, and we can extend this notation to $w \in \Sigma^\star$, and to $R \subseteq Q$. An NFA is *deterministic* (DFA) if for each pair $(q, a) \in Q \times \Sigma$, $|\delta(q, a)| \leq 1$. The *language recognized* by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\star \mid \delta(q_0, w) \cap F \neq \emptyset\}$. The set of languages recognized by NFA's coincides with the set of languages represented by regular expressions, i.e the set of *regular languages*.

Regular languages can be associated to sets of languages equations. Given an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ with $|Q| = n + 1$ we can consider $Q = [0, n]$ and $q_0 = 0$. Let L_i be the language recognized by the automaton $([0, n], \Sigma, \delta, i, F)$, for $i \in [0, n]$ and $\mathcal{L}(\mathcal{A}) = L_0$. Then, the following language equations are satisfied:

$$\begin{array}{ll} L_i &= \left(\bigcup_{j=1}^{k} \{a_j\} L_{ij}\right) \cup \varepsilon(L_i), \quad \forall i \in [0, n], \\ L_{ij} &= \bigcup_{m \in I_{ij}} L_m, \quad I_{ij} = \delta(i, a_j) \subseteq [0, n] \end{array} \tag{1}$$

Conversely any set of languages $\{L_0, \ldots, L_n\}$ that satisfies the set of equations (1) defines an NFA with initial state L_0. In particular if $L_0, \ldots, L_n$ are represented by regular expressions $\alpha \equiv \alpha_0, \ldots, \alpha_n$, respectively, the following holds:

$$\begin{array}{ll} \alpha &\equiv \alpha_0 \\ \alpha_i &= a_1\alpha_{i1} + \ldots + a_k\alpha_{ik} + \varepsilon(\alpha_i), \text{ for } i \in [0, n] \\ \alpha_{ij} &= \sum_{m \in I_{ij}} \alpha_m, \quad I_{ij} \subseteq [0, n] \end{array} \tag{2}$$

Given $\alpha \in \mathsf{RE}$, to find a set of re's that satisfies (2) is tantamount to find an NFA equivalent to α.

3 Partial Derivative Automata

There are several constructions to obtain NFA from re's. Based on the notion of derivative, Brzozowski [Brz64] established a construction of a DFA from a re. The partial derivative automaton ($\mathcal{A}_{pd}$), introduced by V. Antimirov [Ant96], is a non-deterministic version of the Brzozowski automaton.

For a re $\alpha \in \mathsf{RE}$ and a symbol $a \in \Sigma$, the set $\partial_a(\alpha)$ of *partial derivatives* of α w.r.t. a is defined inductively as follows:

$$\begin{array}{ll} \partial_a(\emptyset) &= \partial_a(\varepsilon) = \emptyset \\ \partial_a(b) &= \begin{cases} \{\varepsilon\} \text{ if } b \equiv a \\ \emptyset \quad \text{otherwise} \end{cases} \\ \partial_a(\alpha^\star) &= \partial_a(\alpha) \odot \alpha^\star \end{array} \qquad \begin{array}{ll} \partial_a(\alpha + \beta) &= \partial_a(\alpha) \cup \partial_a(\beta) \\ \partial_a(\alpha\beta) &= \begin{cases} \partial_a(\alpha) \odot \beta \cup \partial_a(\beta) \text{ if } \varepsilon(\alpha) = \varepsilon \\ \partial_a(\alpha) \odot \beta \quad \text{otherwise,} \end{cases} \end{array}$$

where the operator $\odot$ is defined ahead. Let $S \subseteq \mathsf{RE}$ and $\beta \in \mathsf{RE}$. Then $S \odot \beta = \{\alpha\beta \,|\, \alpha \in S\}$ if $\beta \neq \emptyset$, and $S \odot \emptyset = \emptyset$ otherwise. Analogously, one defines $\beta \odot S$. Moreover, if $S = \{\alpha_1, \ldots, \alpha_n\}$, let $\sum S$ denote the re $\alpha_1 + \cdots + \alpha_n$.

Lemma 1 (Antimirov). *For any $a \in \Sigma$ and $\alpha \in \mathsf{RE}$, $\mathcal{L}(\sum \partial_a(\alpha)) = a^{-1}\mathcal{L}(\alpha)$.*

The definition of partial derivative can be extended to sets of regular expressions, words, and languages. Given $\alpha \in \mathsf{RE}$ and $a \in \Sigma$, $\partial_a(S) = \cup_{\alpha \in S} \partial_a(\alpha)$ for $S \subseteq \mathsf{RE}$, $\partial_\varepsilon(\alpha) = \{\alpha\}$, $\partial_{wa}(\alpha) = \partial_a(\partial_w(\alpha))$ for $w \in \Sigma^\star$, and $\partial_L(\alpha) = \cup_{w \in L} \partial_w(\alpha)$ for $L \subseteq \Sigma^\star$. Lemma 1 can be extended to words $w \in \Sigma^\star$. The *set of partial derivatives* of α is defined by $PD(\alpha) = \partial_{\Sigma^\star}\alpha$. An important fact is that $|PD(\alpha)| \leq |\alpha|_\Sigma + 1$. Given a regular expression α, the *partial derivative automaton* $\mathcal{A}_{pd}(\alpha)$ is thus defined as

$$\mathcal{A}_{pd}(\alpha) = (PD(\alpha), \Sigma, \delta_{pd}, \alpha, \{q \in PD(\alpha) \mid \varepsilon(q) = \varepsilon\}),$$

where $\delta_{pd}(q, a) = \partial_a(q)$, for all $q \in PD(\alpha)$ and $a \in \Sigma$.

Proposition 1 (Antimirov). $\mathcal{L}(\mathcal{A}_{pd}(\alpha)) = \mathcal{L}(\alpha)$.

Champarnaud and Ziadi [CZ01] proved that partial derivatives and Mirkin's prebases [Mir66] lead to identical constructions of non-deterministic automata. We now review Mirkin's construction. Given $\alpha \equiv \alpha_0 \in \mathsf{RE}$, the set $\pi(\alpha) = \{\alpha_1, \ldots, \alpha_n\}$, where $\alpha_1, \ldots, \alpha_n$ are non-empty re's, is called a *support* of α if it satisfies the set of equations (2), where α_{ij}, for $i \in [0, n]$ and $j \in [1, k]$, is a summation of elements of $\pi(\alpha)$. If $\pi(\alpha)$ is a support of α, then the set $\pi(\alpha) \cup \{\alpha\}$ is called a *prebase* of α. B. Mirkin provided an algorithm for the computation of a support of a re for which Champarnaud and Ziadi gave an elegant inductive definition[2].

Proposition 2 (Mirkin/Champarnaud&Ziadi). *Let $\alpha \in RE$. Then, the set $\pi(\alpha)$, inductively defined by*

$$\begin{array}{ll} \pi(\emptyset) = \emptyset & \pi(\alpha + \beta) = \pi(\alpha) \cup \pi(\beta) \\ \pi(\varepsilon) = \emptyset & \pi(\alpha\beta) = \pi(\alpha) \odot \beta \cup \pi(\beta) \\ \pi(a) = \{\varepsilon\} & \pi(\alpha^\star) = \pi(\alpha) \odot \alpha^\star, \end{array}$$

is a support of α.

In his original paper Mirkin showed that $|\pi(\alpha)| \leq |\alpha|_\Sigma$. Furthermore, Champarnaud and Ziadi established that $PD(\alpha) = \pi(\alpha) \cup \{\alpha\}$. This fact can be proved noticing that $\mathcal{A}_{pd}(\alpha)$ verifies equations (1) which lead exactly to a language based version of equalities (2) when considering $\alpha_{ij} = \sum \partial_{a_j} \alpha_i$, for $i \in [0, n]$ and $j \in [1, k]$. To prove Proposition 1 is then equivalent to prove Proposition 2. The main result presented in this paper is the formalization of Proposition 2 in Coq.

4 The Coq Proof Assistant

The Coq proof assistant is an implementation of the *Calculus of Inductive Constructions* (CIC) [BC04], a typed λ-calculus that features polymorphism, dependent types and very expressive (co-)inductive types. Coq provides users with the means to define data-structures and functions, as in standard functional languages, and also allows to define specifications and to build proofs in the same language, if we consider the underlying λ-calculus as an higher-order logic under the *Curry-Horward isomorphism* programs-as-proofs principle (CHi) [How80].

In CHi, any typing relation $t : A$ can either be seen as a value t of type A, or as t being a proof of the proposition A. Any type in Coq is in the set of *sorts* $\mathcal{S} = \{\mathsf{Prop}\} \cup \{\mathsf{Type}(i) \mid i \in \mathbb{N}\}$. The $\mathsf{Type}(0)$ sort represents computational types, while the Prop type represents logical propositions.

An inductive type is introduced by a collection of *constructors*, each with its own arity. A value of an inductive type is a composition of such constructors. As an example, natural numbers are encoded as follows:

```
Inductive ℕ : Type :=
| 0 : ℕ | S : ℕ → ℕ.
```

[2] That definition was corrected by Broda *et al.* [BMMR10].

Coq automatically generates induction and recursion principles for each new inductive type. More complex types can be created such as *dependent types*. As an example of a dependent type, consider *subset types* (or Σ-types), formalized in Coq as the type sig[3].

```
Inductive sig (A:Type) (P:A → Prop) : Type :=
| exist : ∀ x:A, P x → sig P.
```

The type sig (that has a syntactical notation of { x | P}) is pair (x, H), where x represents some computational value and H has the type $P(x)$ and is a proof that x satisfies P. Assuming the predicate Even x which encodes even natural numbers, an example of a Σ-type is the subset-type of even naturals sig Even, and one of its possible inhabitants is the term value exist 2 (Even 2). The definition of nested subset types is provided the following dependent type

```
Inductive sigS (A:Type) (P:A → Type) : Type :=
| existS : ∀ x:A, P x → sigS A P.
```

(with the syntactical notation {x:A & P}), where P is either a sig or a sigS type.

In Coq, functions must be provably terminating. Termination is ensured by a guard predicate that checks that recursive calls are always performed on structurally smaller arguments. As an example, consider the function plus that adds two natural numbers.

```
Fixpoint plus (n m:ℕ) {struct n} : ℕ :=
 match n with
 | 0 ⇒ m | S p ⇒ S (plus p m)
 end.
```

The basic way of the Coq proof construction process is to explicitly build CIC terms. However, proofs can be built more conveniently and interactively in a backward fashion. This step by step process is done by the use of *proof tactics*.

Another appealing feature of Coq is the possibility to extract the constructive parts of proof development into correct by construction functional programs. Since the underlying logic of Coq is constructive, any value, proof included, can be seen as a (functional) program. The extraction mechanism keeps the computational counterparts and translate them into standard functional programs. On the other hand, purely logical sub-terms are discarded since they are computationally non-informative.

In this paper we use the Coq libraries Ensembles and FSets that formalize sets. The Ensembles library formalizes the notion of set as a characteristic predicate. The base type is Ensemble (X:Type) := X → Prop. Set operations are also provided. As an example, consider the *singleton* and the *union*:

```
Definition In(U:Type)(P:Ensemble U)(x:U) := P x.

Inductive Singleton(U:Type)(x:U) : Ensemble U :=
| In_singleton : In U (Singleton x) x.

Inductive Union(U:Type)(B C:Ensemble U) : Ensemble U :=
| Union_introl : ∀ x:U, In U B x → In U (Union B C) x
| Union_intror : ∀ x:U, In U C x → In U (Union B C) x.
```

[3] Note that the second argument of sig is a proof term that depends on the first argument.

The FSets library provides a rich implementation of finite sets over decidable and/or ordered types.

5 Formalization in Coq

This section describes the main parts of our formalization in Coq. First we present the formalization of regular languages and re's.

5.1 Formal Languages and Regular Expressions

An alphabet sigma (Σ) can be specified as a non-empty list of symbols of a type A. It is required that the type A is ordered. For that, a proof of compare_sy must be given, that is, a term of type Compare and that corresponds to a function that receives two variables x, y of type A and that returns a term proving if x and y are either equal, or if $x < y$, or $y < x$.

```
Inductive Compare (A : Type) (lt eq : A → A → Prop) (x y : A) : Type :=
| LT : lt x y → Compare lt eq x y
| EQ : eq x y → Compare lt eq x y
| GT : lt y x → Compare lt eq x y.

Module Type Alphabet.
 Parameter  A : Set.
 Definition A_eq := (eq A).
 Parameter  A_lt : A → A → Prop.
 Parameter  sigma : list A.
 Axiom sigma_nempty : sigma ≠ nil.
 Axiom compare_sy : ∀ x y:A, Compare sylt syeq x y.
End Alphabet.
```

Words are lists whose elements have type A, and that belong to sigma. A word w is a valid word if $w \in \Sigma^\star$ which correspond to the IsWord predicate.

```
Definition IsSy(a:A) := a ∈ sigma.
Definition word := list A.

Inductive IsWord : word → Prop :=
| nil_IsWord  : IsWord ε
| cons_IsWord : ∀ a:A, IsSy a → ∀ u:word, IsWord u → IsWord (a::u).
```

Languages are sets of words, that is, terms of type Ensemble word. The languages $\emptyset$, $\{\epsilon\}$, $\{a\}$ for $a \in \Sigma$, and language union are defined using the corresponding Ensembles definitions. Concatenation and Kleene's star are formalized as the predicates $\cdot$ and * as presented below. Equivalence of languages is denoted by $=_{\mathcal{L}}$ which is the standard set equivalence, and it is represented by the predicate Same_set.

```
Definition language := Ensemble word.

Definition ∅ := (Empty word).
Definition ε := (Singleton word nil).
Definition ([S] x) := (Singleton word (x::nil)).
Definition (x ∪ y) := Union word x y.

Inductive (L1 · L2:language) : language :=
 | ConcL_app : ∀ w1 w2:word, w1 ∈ L1 → w2 ∈ L2 → (w1 ++ w2) ∈ (L1 · L2).
```

```
Fixpoint lpow (L:language)(n:ℕ) : language :=
 match n with
 |0 => ε  | (S m) => (L · (lpow L m))
 end.

Inductive (L:language)* : language :=
| starL_n : ∀ n:ℕ w, w ∈(lpow L n) → w ∈(L*).

Definition (L1 =ℒ L2:language) := (Same_set L1 L2).
```

Several properties of regular languages were proved, and, in particular, that regular languages are a model for KA. This was accomplished considering the KA implementation in Coq described in [PM08]. An extended description of that proof is presented in [MPdS09]. Regular expressions are encoded by the inductive type re. The language of any re α is obtained by applying the function re2rel to α. This function was proved correct w.r.t. to RL, the predicate that defines regular languages over the alphabet sigma (Theorem re2rel_is_RL).

```
Inductive re : Set :=
| re0 : re | re1 : re | re_sy : ∀ a:A, IsSy a → re
| re_union : re → re → re | re_conc : re → re → re
| re_star : re → re.

Fixpoint re2rel (α:re) : language :=
 match x with
 | re0 ⇒ ∅ | re1 ⇒ ε | re_sy a H ⇒ ([S] a)
 | re_union α1 α2 ⇒ (re2rel α1) ∪ (re2rel α2)
 | re_conc α1 α2 ⇒ (re2rel α1) · (re2rel α2)
 | re_star α1 ⇒ (re2rel α1) [*]
 end.

Coercion re2rel : re >-> language.

Inductive RL : language → Prop :=
| RL0 : RL ∅ | RL1 : RL ε | RLs : ∀ a, RL ([S] a)
| RLp : ∀ l1 l2, RL l1 → RL l2 → RL (l1 ∪ l2)
| RLc : ∀ l1 l2, RL l1 → RL l2 → RL (l1 · l2)
| RLst : ∀ l, RL l → RL (l*).

Theorem re2rel_is_RL : ∀ α:re, RL α.
```

In the code above, re2rel was declared as a *coercion* which allows one to refer to a given re α where its language is required, i.e., without explicitly referring to re2rel(α). For instance, the concatenation of the languages corresponding to the re's α_1 and α_2, is written $\alpha_1 \cdot \alpha_2$ instead of (re2rel(α_1))·(re2rel (α_2)).

5.2 Correctness of Mirkin's Construction

We now present the formalization of our main result, i.e., given a re α, $\pi(\alpha)$ computes a support for α.

The function π is formalized in Coq as a structural recursive function, and thus its termination is ensured. The function _ ⊙ _ is defined using the auxiliary function fold_conc, which concatenates a re to the right of each element of a set of re's. A set of re's is represented by the type re_set).

```
Definition fold_conc (s:re_set)(r:re) :=
  fold (fun x ⇒ add (re_conc x r)) s empty.

Definition ⊙ (s:re_set)(r:re):t :=
```

```
  match r with
  | re0 ⇒ empty | _ ⇒ fold_conc s r
  end.

Fixpoint π (r:re) : re_set :=
 match r with
 | re0 ⇒ ∅ | re1 ⇒ ∅ | re_sy _ _ ⇒ {re1}
 | re_union x y ⇒ (π x) ∪ (π y)
 | re_conc x y  ⇒ ((π x)⊙ y) ∪ (π y)
 | re_star x ⇒ ⊙ (π x) (re_star x)
end.
```

The proof that $\pi(\alpha) \leq |\alpha|_\Sigma$ is provided by Theorem PI_upper_bound, where the function $|\_|_\Sigma$ is defined by structural induction.

```
Fixpoint |α:re|Σ : ℕ :=
 match r with
 | re0 ⇒ 0 | re1 ⇒ 0 | re_sy s _ ⇒ 1
 | re_union x y ⇒ (|x|Σ) + (|y|Σ)
 | re_conc x y ⇒ (|x|Σ) + (|y|Σ)
 | re_star x ⇒ |x|Σ
 end.

Lemma PI_upper_bound : ∀ r:re, cardinal(π r) ≤ |r|Σ.
```

Recall that the function π is a support if its elements satisfy the equations of (2). The set of equations (2) is defined by the inductive type Support, which has two constructors.

```
Inductive Support (r:re)(s:re_set) : language :=
| mb_empty : ∀ w:word,  w ∈ ε(r) → w ∈(Support r s)
| mb :  ∀ (w:word) (a:sy) (H:IsSy a),  ¬Empty s →
   {x:re & {s':re_set | x ∈ s ∧ w ∈ ((re_sy a H) · x) ∧
        ((re_sy a H̄) · x) ⊆r ∧ s' ⊆ s ∧ x =ℒ LS s')}} →
           w ∈ (Support r s).
```

The first constructor, mb_empty, corresponds to the case where a word w belongs to $\varepsilon(r)$. The second constructor, mb, corresponds to the case where a word w belongs to one of the other parcels of the summation in the right side of equations (2). It has a Σ-type as argument which is a witness, $(x, (s', P))$, that $P\,x\,s'$ is a proof that for each $a \in \Sigma$, the parcel $a \cdot \alpha_{il}$ is such that α_{il} is built from an $s' \subseteq s$. The argument ¬Empty s is introduced for technical reasons only and to facilitate proof construction.

Our main result is the proof that π calculates the support for a given re. This is established in the following theorem:

```
Theorem PI_is_MSupport : ∀ r:re, r =ℒ MSupport r (π r).
```

The proof of this theorem follows the original proof provided by Mirkin. The proof is constructed by induction on α. The equivalence between partial derivative automata and Mirkin's construction was also proved correct through the following theorem:

```
Lemma SupportParts_are_pdrv :  ∀ r w,
  w ?? (IsSupport r (PI r)) → w = nil \/ exists a, w ?? ([S a] [C] (pdrv r a)).
```

The above theorem is a formalization of the result established by Champarnaud and Ziadi [CZ01] where they show that the elements of the support correspond to partial derivatives.

6 Related Work and Applications

Formalization of finite automata in Coq was first approached by J.-C. Filliâtre in [Fil97]. The author's aim was to prove the *pumping lemma* for regular languages and the extraction of an OCaml program. More recently, S. Briais [Bri08] developed a new formalization of formal languages in Coq, which covers Filliâtre's work. This formalization includes Thompson construction of an automaton from a re and a naïve construction of two automata equivalence based in testing if the difference of their languages is the empty language. Braibant and Pous [BP09] formalized KA based on Kozen's algebraic proof of completeness of KA, and provided reflexive tactics to automatically decide KA expression's equivalence. Benoît Razet [Raz08] formalized an executable complete simulator for finite Eilenberg machines. The soundness of the corresponding algorithm was proved sound and correct, and was extracted as an executable OCaml program. Pereira and Moreira [PM08] also presented a formalization of KA and KAT in Coq. Kleene algebra with tests (KAT) [Koz97] extends KA with an embedded *Boolean algebra* and it is particularly suited for the formal verification of propositional programs. In particular, KAT subsumes *propositional Hoare logic* (PHL) [KT00], a Hoare logic without the assignment axiom. Pereira and Moreira provided a mechanically verified proof that the deductive rules of PHL are theorems of KAT. However, no automation mechanisms were considered.

7 Concluding Remarks

In this paper we have described a formalization of regular languages in the Coq proof assistant. Our main result is the correctness of Mirkin's construction of partial derivative automaton from a regular expression. The overall formalization consists of approximately 2700 lines of specification code, and approximately 7900 lines of proof code. The results of this paper provide the base for the correctness of a decision procedure for re equivalence, based on the notion of derivative. This kind of procedure was presented by several authors [Brz64, AM95, AMR09a, AMR09b].

This work is the continuation of previous work on the formalization of KA, KAT, and PHL in Coq. Since Coq is a proof assistant that allows the compilation of proofs into binary proof objects, we envision the representation of propositional programs and their properties in the context of Proof-Carrying-Code [Nec97]. In this context programs are packaged together with the certificates that assert program partial correctness.

Acknowledgments

We thank Sebastien Briais for his willingness to respond promptly to questions concerning his implementation of formal languages in the Coq proof assistant. This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI and the project RESCUE (PTDC/EIA/65862/2006). David Pereira is funded by FCT grant SFRH/BD/33233/2007.

References

[AM95] Antimirov, V.M., Mosses, P.D.: Rewriting extended regular expressions. Theor. Comput. Sci. 143(1), 51–72 (1995)

[AMR09a] Almeida, M., Moreira, N., Reis, R.: Antimirov and Mosses's rewrite system revisited. International Journal Of Foundations Of Computer Science 20(04), 669–684 (2009)

[AMR09b] Almeida, M., Moreira, N., Reis, R.: Testing equivalence of regular languages. In: DCFS 2009, Magdeburg, Germany (2009)

[Ant96] Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. Theoret. Comput. Sci. 155(2), 291–319 (1996)

[BC04] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science. Springer, Heidelberg (2004)

[BMMR10] Broda, S., Machiavelo, A., Moreira, N., Reis, R.: On the average number of states of partial derivative automata. In: Gao, Y., Lu, H., Seki, S., Yu, S. (eds.) DLT 2010. LNCS, vol. 6224, pp. 112–123. Springer, Heidelberg (2010)

[BP09] Braibant, T., Pous, D.: A tactic for deciding Kleene algebras. In: First Coq Workshop (Available as a HAL report) (August 2009)

[Bri08] Briais, S.: Finite automata theory in Coq (2008), http://www.prism.uvsq.fr/~bris/tools/Automata_080708.tar.gz

[Brz64] Brzozowski, J.A.: Derivatives of regular expressions. JACM 11(4), 481–494 (1964)

[CZ01] Champarnaud, J.M., Ziadi, D.: From Mirkin's prebases to Antimirov's word partial derivatives. Fundam. Inform. 45(3), 195–205 (2001)

[Fil97] Filliâtre, J.-C.: Finite automata theory in Coq - a constructive proof of Kleene's theorem (1997)

[How80] Howard, W.A.: The formulae-as-types notion of construction, pp. 479–490. Academic Press, London (1980)

[Kle56] Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Shannon, C.E., McCarthy, J. (eds.) Automata Studies, pp. 3–41. Princeton University Press, Princeton (1956)

[Koz94] Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Infor. and Comput. 110(2), 366–390 (1994)

[Koz97] Kozen, D.: Kleene algebra with tests. Transactions on Programming Languages and Systems 19(3), 427–443 (1997)

[KT00] Kozen, D., Tiuryn, J.: On the completeness of propositional Hoare logic. In: RelMiCS, pp. 195–202 (2000)

[Mir66] Mirkin, B.G.: An algorithm for constructing a base in a language of regular expressions. Engineering Cybernetics 5, 51–57 (1966)

[MPdS09] Moreira, N., Pereira, D., de Sousa, S.M.: On the mechanization of Kleene algebra in Coq. Technical Report DCC-2009-03, DCC-FC&LIACC, Universidade do Porto (2009)

[Nec97] Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages POPL 1997, pp. 106–119. ACM, New York (1997)

[PM08] Pereira, D., Moreira, N.: KAT and PHL in Coq. Computer Science and Information Systems 05(02) (December 2008) ISSN: 1820-0214

[Raz08] Razet, B.: Simulating Eilenberg Machines with a Reactive Engine: Formal Specification, Proof and Program Extraction. Research Report (2008)

Regular Geometrical Languages and Tiling the Plane

Jean-Marc Champarnaud, Jean-Philippe Dubernard, and Hadrien Jeanne

LITIS, University of Rouen, France
{jean-marc.champarnaud,jean-philippe.dubernard}@univ-rouen.fr,
hadrien.jeanne@univ-rouen.fr

Abstract. We show that if a binary language L is regular, prolongable and geometrical, then it can generate, on certain assumptions, a $p1$ type tiling of a part of $\mathbb{N}^2$. We also show that the sequence of states that appear along a horizontal line in such a tiling only depends on the shape of the tiling sub-figure and is somehow periodic.

1 Introduction

A tiling of the plane is a finite set of plane figures, called tiles, instances of which fill the plane with no overlaps and no gaps. A $p1$ type tiling uses only translated instances. A polyomino [8] is a polygon exactly covered by unit squares whose edges are horizontal or vertical. Numerous decidability results have been published addressing the problem of tiling the plane with polyominoes; see for example [2,11] for tiling with one polyomino. A language is geometrical [3] if the set of its prefixes can be drawn over $\mathbb{N}^2$ and if this set is equal to the language of the resulting geometrical figure. The study of geometrical languages was initially motivated by their application to the modeling of real-time task systems [1], via regular languages [7] or discrete geometry [10,7]. An automaton-based characterization has been provided for the family of regular geometrical binary languages [5,4]. In this paper, we make use of this characterization and show that, on certain hypotheses, a $p1$ type tiling of a part of $\mathbb{N}^2$ can be generated by such a language. The following two sections recall fundamental notions concerning languages, finite automata and 2-dimensional geometrical languages. Section 4 investigates the conditions for the figure of a geometrical language to admit a region that is tiled by a specific sub-figure. In Section 5 it is shown that, given such a tiling, the sequence of states that appear along a horizontal line only depends on the shape of the tiling sub-figure and is in some way periodic.

2 Preliminaries

Let us first review basic notions concerning regular languages and finite automata. For a comprehensive treatment of this domain, reference [6] can be consulted. Let Σ be a nonempty finite set of symbols, called the *alphabet*. A *word* over Σ is a finite sequence of symbols, usually written $x_1x_2\cdots x_n$. The *length* of a word u, denoted by $|u|$, is the number of symbols in u. The number of occurrences of a symbol a in u is denoted by $|u|_a$. The *empty word*, denoted by ε, has a null length. If $u = x_1\cdots x_n$ and

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 69–78, 2011.

$v = y_1 \cdots y_m$ are two words over the alphabet Σ, their concatenation $u \cdot v$, usually written uv, is the word $x_1 \cdots x_n y_1 \cdots y_m$. Let Σ^* be the set of words over Σ. Given two words u and w in Σ^*, u is said to be a *prefix* of w if there exists a word v in Σ^* such that $uv = w$. A *language* L over Σ is a subset of Σ^*. The set of prefixes of the words of the language L is denoted by $\mathrm{Pref}(L)$. The set of *regular* languages over an alphabet Σ contains the empty set and the set $\{a\}$ for all $a \in \Sigma$ and it is closed under finite concatenation, finite union and star. A language L is said to be *prolongable* if and only if for all u in L, there exists x in Σ such that $u \cdot x \in L$. Let L be a language over the alphabet Σ and u be a word in Σ^*; the *left quotient* of L by u is the set $u^{-1}(L) = \{v \in \Sigma^* \mid uv \in L\}$.

A *deterministic finite automaton* (DFA) is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ where Q is a finite *set of states*, δ is a mapping from $Q \times \Sigma$ to Q, s_0 is the *initial state* and T is the *set of final states*. For all $(p, x) \in Q \times \Sigma$, we will write $p \cdot x$ instead of $\delta(p, x)$; the 3-tuple (p, x, q) in $Q \times \Sigma \times Q$ is said to be *a transition* if and only if $q = p \cdot x$. A DFA $\mathcal{A}$ is said to be *complete* if for any $q \in Q$ and any $a \in \Sigma$, $|q \cdot a| = 1$. In a complete DFA there may exist a *sink state* σ such that $\sigma \notin T$ and, for all $x \in \Sigma$, $\sigma \cdot x = \sigma$. Let $p \in Q$ and $u = u_1 \cdots u_l \in \Sigma^*$. The *path* (p, u) of length l starting from p and labeled by u is the sequence of transitions $((p_0, u_1, p_1), \ldots, (p_{l-1}, u_l, p_l))$, with $p_0 = p$. A path (p, u) is said to be *proper* if $p \cdot u \neq \sigma$. It is said to be *successful* if $p = s_0$ and $p \cdot u \in T$. The language $L(\mathcal{A})$ *recognized* by the DFA $\mathcal{A}$ is the set of words that are labels of successful paths. Kleene's theorem [9] states that a language is recognized by a finite automaton if and only if it is regular. The *left language* $\overleftarrow{L}_q^{\mathcal{A}}$ (resp. *right language* $\overrightarrow{L}_q^{\mathcal{A}}$) of a state q is the set of words w such that there exists a path in $\mathcal{A}$ from s_0 to the state q (resp. from q to a final state) with w as label. A DFA $\mathcal{A}$ is said to be *accessible* if for any $q \in Q$ there exists a path from s_0 to q. A complete and accessible DFA $\mathcal{A}$ is *minimal* if and only if any two distinct states of $\mathcal{A}$ have distinct right languages. According to the theorem of Myhill-Nerode [12,13], the minimal DFA of a regular language is unique up to an isomorphism. Notice that if $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ is the minimal automaton of $\mathrm{Pref}(L)$ and if $L \neq \Sigma^*$, then there exists a sink state σ and $Q = T \cup \{\sigma\}$; in this case, any proper path is successful and reciprocally.

A *permutation* φ of a set E of cardinality m is a bijection from E to E. Without loss of generality, we consider that $E = \{0, 1, \ldots, m-1\}$. Two elements i and j of E belong to the same *orbit* if and only if there exists an integer p, $0 \leq p < m$, such that $\varphi^p(i) = j$. We will say that a permutation φ is *circular* if there exists an integer a, $0 \leq a < m$, called *a shift* and such that for all h, $0 \leq h < m$, it holds $\varphi(h) \equiv h + a \mod m$. In this case, i and j belong to the same orbit if and only if the congruence $i + ax \equiv j \mod m$ admits at least one solution.

Lemma 1. *Let φ be a circular permutation of a set E of cardinality m and let a be its shift. Let $d = \gcd(a, m)$. Then the permutation φ admits d orbits $O_0, \ldots, O_{d-1}$, of cardinality m/d. For all r, $0 \leq r < d$, it holds $O_r = (r + kd)_{0 \leq k < m/d}$.*

3 Geometrical Languages

Let us now review basic definitions and properties of geometrical figures and languages, as introduced in [3]. Since this paper deals with binary languages, we only describe the

2-dimensional case. Let $\Sigma = \{a_1, a_2\}$ be a binary alphabet. The Parikh mapping [14] $c : \Sigma^* \longrightarrow \mathbb{N}^2$ maps a word w to its coordinate vector $(|w|_{a_1}, |w|_{a_2})$. In particular, $c(a_1) = (1, 0)$ and $c(a_2) = (0, 1)$. Let $\mathcal{O}$ be the point with coordinate $(0, 0)$. For any point P in $\mathbb{N}^2$, we write P instead of $\overrightarrow{OP}$. The *level* of the point $P = (x_1, x_2)$ is $\text{level}(P) = x_1 + x_2$.

Let F be a subset of $\mathbb{N}^2$ and P be a point in F. A *trajectory* of length l of F starting from P is a sequence $T = (P_i)_{1 \leq i \leq l}$ of points of F, such that $P_0 = P$ and for all i, $1 \leqslant i \leqslant l$, there exists an integer k_i, $1 \leqslant k_i \leqslant 2$, such that $P_i = P_{i-1} + c(a_{k_i})$. Notice that if there exists such an integer k_i, then it is unique, since the coordinate vector of a point is unique. Hence the sequence T is defined by a unique word $u = a_{k_1} \cdots a_{k_l}$ and a trajectory is either a point sequence T or a pair (P, u) in $F \times \Sigma^*$. The set of points of a trajectory (P, u) is denoted by $\text{points}(P, u)$. The *set of trajectories* of F starting from P is denoted by $\text{Traj}(P, F)$. Let P and P' be two points of F; P' is said to be *accessible* from P if and only if it belongs to a trajectory starting from P.

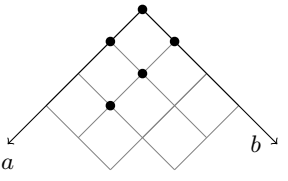


Fig. 1. $F_1 = \{(0, 0), (0, 1), (1, 0), (1, 1), (2, 1)\}$

Definition 1. *A 2-dimensional geometrical figure F is a (possibly empty) subset of $\mathbb{N}^2$ every point of which is accessible from $\mathcal{O}$.*

Figure 1 represents a 2-dimensional geometrical figure, with $\Sigma = \{a, b\}$ as alphabet. Notice that geometrical figures are drawn so that points with the same level lie on a horizontal line. The *geometrical figure of a language* L is defined by $\mathcal{F}(L) = \bigcup_{w \in \text{Pref}(L)} \text{points}(\mathcal{O}, w)$, and the *language of a geometrical figure* F by $\mathcal{L}(F) = \{u \mid (\mathcal{O}, u) \in \text{Traj}(\mathcal{O}, F)\}$. A language L is said to be *geometrical* if and only if $\text{Pref}(L) = \mathcal{L}(\mathcal{F}(L))$. For any language L, $\text{Pref}(L) \subseteq \mathcal{L}(\mathcal{F}(L))$. Some languages however are such that $\mathcal{L}(\mathcal{F}(L)) \not\subseteq \text{Pref}(L)$. For instance, the two languages $\{a, ba\}$ and $\{ab, ba\}$ have the same geometrical figure; the former one is not geometrical, whereas the latter one is.

Let us introduce informally the main properties of geometrical languages. References [3,5] can be consulted for complete proofs. The trajectories of a geometrical figure F have two properties: the intersection property P_1 and the neighbourhood property P_2. The trajectories $(\mathcal{O}, u)$ and $(\mathcal{O}, v)$ are supposed to belong to $\text{Traj}(\mathcal{O}, F)$.

$$P_1\text{: } c(u) = c(v) \Rightarrow \forall w \in \Sigma^*, (\mathcal{O}, uw) \in \text{Traj}(\mathcal{O}, F) \Leftrightarrow (\mathcal{O}, vw) \in \text{Traj}(\mathcal{O}, F)$$
$$P_2\text{: } c(ub) = c(va) \Rightarrow (\mathcal{O}, ub) \in \text{Traj}(\mathcal{O}, F) \Leftrightarrow (\mathcal{O}, va) \in \text{Traj}(\mathcal{O}, F)$$

The corresponding properties of the language $L(F)$ of a geometrical figure are:

$$P_1'\text{: } c(u) = c(v) \Rightarrow \forall w \in \Sigma^*, (uw \in L(F) \Leftrightarrow vw \in L(F))$$
$$P_2'\text{: } c(ub) = c(va) \Rightarrow (ub \in L(F) \Leftrightarrow va \in L(F))$$

The geometrical figure $F = \mathcal{F}(L)$ of a language L is built by drawing the trajectories of the words of L over $\mathbb{N}^2$. According to properties P_1 and P_2, new trajectories can appear and thus it should be checked that the corresponding words in the language $\mathcal{L}(\mathcal{F}(L))$ also exist in $\mathrm{Pref}(L)$. The two following conditions are necessary for the equality $\mathrm{Pref}(L) = \mathcal{L}(\mathcal{F}(L))$ to be satisfied. They apply to any language, not necessarily a regular one. The words u and v are supposed to belong to $\mathrm{Pref}(L)$.

The *semi-geometricity* condition, C_1, is a consequence of the property P'_1.

$$C_1 : c(u) = c(v) \Rightarrow u^{-1}\mathrm{Pref}(L) = v^{-1}\mathrm{Pref}(L)$$

The *neighbourhood* condition, C_2, is a consequence of the property P'_2.

$$C_2 : c(ub) = c(va) \Rightarrow (ub \in \mathrm{Pref}(L) \Leftrightarrow va \in \mathrm{Pref}(L))$$

It can be checked that $C_1 \Leftarrow \mathrm{Pref}(L) = \mathcal{L}(\mathcal{F}(L))$ and $C_2 \Leftarrow \mathrm{Pref}(L) = \mathcal{L}(\mathcal{F}(L))$.

For example, let $L_0 = \{aba, b\}$, $L_1 = \{aba, ba\}$ and $L_2 = \{aba, baa\}$ be three languages that admit the same geometrical figure F_0 represented by Figure 1. The language L_0 satisfies C_1 but does not satisfy C_2, whereas L_1 satisfies C_2 but does not satisfy C_1. The languages L_0 and L_2 are semi-geometrical. The language L_2 is geometrical.

It can be proved[1] that the *geometricity condition* $C = C_1 \wedge C_2$ is a necessary and sufficient condition for the equality $\mathrm{Pref}(L) = \mathcal{L}(\mathcal{F}(L))$ to be satisfied.

We now suppose that L is a regular language and we rewrite conditions C_1 and C from the minimal automaton $\mathcal{A}$ of $\mathrm{Pref}(L)$. Let u and v be two words of $\mathrm{Pref}(L)$. Since $\mathcal{A}$ is the minimal automaton of $\mathrm{Pref}(L)$, it holds $u^{-1}\mathrm{Pref}(L) = v^{-1}\mathrm{Pref}(L) \Leftrightarrow s_0 \cdot u = s_0 \cdot v$. Hence the condition C_1 can be rewritten C'_a or C'_b:

$$C'_a : \forall u \in \mathrm{Pref}(L), \forall v \in \mathrm{Pref}(L), c(u) = c(v) \Rightarrow s_0 \cdot u = s_0 \cdot v$$
$$C'_b : \forall P \in \mathcal{F}(L), \exists! p \in Q \mid (\forall u \in \mathrm{Pref}(L), c(u) = P \Leftrightarrow s_0 \cdot u = p)$$

Let State be the mapping from $\mathcal{F}(L)$ to Q such that for all $P \in \mathcal{F}(L)$, $\mathrm{State}(P)$ is the unique state satisfying the condition C'_b. For all $P \in \mathbb{N}^2 \setminus \mathcal{F}(L)$, since the condition C'_b is satisfied by the state σ, we can set $\mathrm{State}(P) = \sigma$. Let $p_b = \mathrm{State}(P - c(b))$ and $p_a = \mathrm{State}(P - c(a))$. Finally, the geometricity condition C can be rewritten:

$$C' : \forall (P, P - c(a), P - c(b)) \in \mathbb{N}^2 \times \mathcal{F}(L) \times \mathcal{F}(L), p_a \cdot a = p_b \cdot b = \mathrm{State}(P)$$

The *basic figure* F_Q of a geometrical figure $\mathcal{F}(L)$ is the subset of points P of $\mathcal{F}(L)$ such that $\forall P' \in \mathcal{F}(L), \mathrm{State}(P) = \mathrm{State}(P') \Rightarrow (x + y < x' + y') \vee ((x + y = x' + y') \wedge (x > x'))$, where x and y (resp. x' and y') are the coordinate of P (resp. of P'). The restriction of State to F_Q is a bijection. The inverse mapping is the mapping Point from Q to F_Q.

4 Geometrical Languages and Tiling

We first investigate the properties of some sub-figures of an arbitrary geometrical figure F. We then consider a regular prolongable binary language L. Assuming that L is geometrical, we focus on the properties of the paths of the minimal DFA of $\mathrm{Pref}(L)$. We then state a set of conditions for L to define a partial tiling of $\mathbb{N}^2$.

[1] The wording of condition C_2 is slightly different in [3].

4.1 Tiling a Geometrical Figure

Let F be a geometrical figure and A be a point of F. Let z and z' be two distinct words of Σ^+ such that $c(z) = c(z')$. We suppose that (A, z) and (A, z') are two trajectories of F. These trajectories are the *boundary* of a *sub-figure* of F called *a quasi-polygon* and denoted by (A, z, z'). The interior of a quasi-polygon may contain one or several points of F; otherwise the quasi-polygon is *empty*.

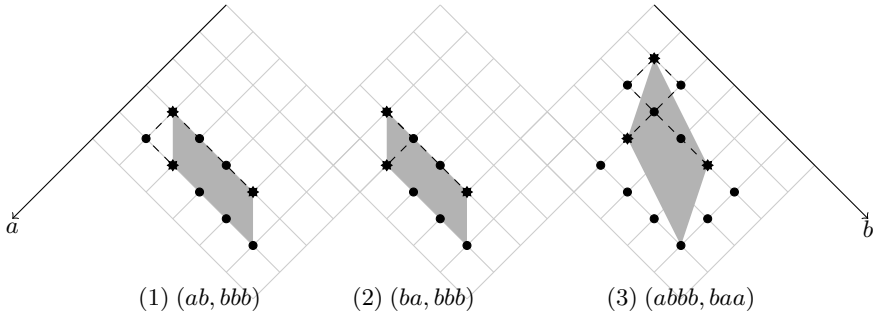


Fig. 2. Quasi-parallelograms $(A, ss', s's)$ of $\mathbb{N}^2$

Let s and s' be two distinct words of Σ^+. The quasi-polygon $(A, ss', s's)$ is called *a quasi-parallelogram*. The notion of quasi-parallelogram is illustrated by Figure 2. Let $B = A + c(s)$, $D = A + c(s')$ and $C = B + c(s') = D + c(s)$. The opposite sides of the parallelogram $ABCD$ (resp. of the quasi-parallelogram $ABCD$) are pairwise parallel segments (resp. trajectories). The boundary of a quasi-parallelogram $(A, ss', s's)$ is obtained by deforming the associated parallelogram $ABCD$. Indeed, the parallel sides AB and CD of the parallelogram are identically deformed, by action of the word s, yielding the trajectories (A, s) and (C, s) of the quasi-parallelogram. Similarly, the sides AC and BD yield, by action of the word s', the trajectories (A, s') and (B, s'). Consequently, the boundary of $(A, ss', s's)$ may define a $p1$ type tiling in the figure F. It is the case for example if, for all $\alpha \geq 1$ and $\beta \geq 1$, the trajectories $(A, s^\alpha s'^\omega)$ and $(A, s'^\beta s^\omega)$ are trajectories of F.

By definition, a quasi-parallelogram is a set of points since it is a sub-figure of a geometrical figure. An empty quasi-parallelogram can also be viewed as a particular parallelogram polyomino. It is proved in [2] that tilings of the plane by translation are generated by specific polyominoes called exact polyominoes. It can be easily checked that an empty quasi-parallelogram defines an exact polyomino. A first difference from previous works dealing with tiling of the plane by polyominoes is that quasi-parallelograms occurring in geometrical figures are not necessarily empty.

Let us assume that the quasi-parallelogram $(A, ss', s's)$ contains $n_i \geq 0$ interior points. For all interior point M_i, there exists a word $w_i \in \Sigma^+$ such that $C = A + c(w_i) \wedge (A, w_i) \in T(A, F) \wedge M_i \in (A, w_i)$. Thus it is sufficient that, for all $\alpha \geq 1$, $\beta \geq 1$ and $0 \leq i \leq n_i$, the trajectories $(A, s^\alpha w_i^\omega)$ and $(A, s'^\beta w_i^\omega)$ also be trajectories of F for the quasi-parallelogram to induce a tiling.

A quasi-parallelogram is said to be *tiling* if it induces a tiling of F. Let $(A, ss', s's)$ be a tiling quasi-parallelogram. Then the trajectory (A, s^ω) is in $T(A, F)$ and it is called *the quasi-axis* X_s. Similarly, the trajectory (A, s'^ω) is called *the quasi-axis* $Y_{s'}$. The region of $\mathbb{N}^2$ delimited by X_s and $Y_{s'}$ is *the* (s, s')*-cone* with A as origin. The tiling by $(A, ss', s's)$ covers the (s, s')-cone with A as origin.

4.2 Tiling the Figure of a Geometrical Language

In this section we consider a regular prolongable binary language L and we assume that L is geometrical. Let $\mathcal{F}(L)$ be the geometrical figure of L and $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ be the minimal DFA of $\mathrm{Pref}(L)$. We investigate the connections between the properties of the paths of $\mathcal{A}$ and those of the sub-figures of $\mathcal{F}(L)$.

By definition, a path of an automaton is a transition sequence. In the following we will consider specific proper paths (p, u), called *state paths*, such that for all $0 \leq i < |u|$, the label u_{i+1} of the transition (p_i, u_{i+1}, p_{i+1}) only depends on the state p_i. A state path (p, u) amounts to the state sequence $(p_0, \ldots, p_{|u|})$. The *right path* (p, u) of a state $p \in T$ is the state path defined by the word $u = u_1 \cdots$ such that $u_{i+1} = b$ **if** $p \cdot (u_1 \cdots u_i \cdot b) \neq \sigma$ and $u_{i+1} = a$ **otherwise**. Since L is prolongable, it holds $p \cdot (u_1 \cdots u_i \cdot b) = \sigma \Rightarrow p \cdot (u_1 \cdots u_i \cdot a) \neq \sigma$ and consequently $p \cdot (u_1 \cdots u_{i+1}) \neq \sigma$. The *left path* (p, u') of p is defined by the word $u' = u'_1 \cdots$ such that $u'_{i+1} = a$ **if** $p \cdot (u'_1 \cdots u'_i \cdot a) \neq \sigma$ and $u'_{i+1} = b$ **otherwise**.

A state path (p, u) is *periodic* if there exist two integers r and s, $0 \leq r < s \leq |u|$, such that $p_r = p_s$. We assume that r and s are the smallest integers satisfying this equality. Then there exist two words $v \in \Sigma^*$ and $w \in \Sigma^+$ such that $|v| + |w| < n$ and $p \cdot v = p \cdot vw = p_r$, with $u = vw^\omega$. The path is said to have a *pre-period* $|v|$ and a *period* $|w|$. The language L being by hypothesis prolongable, every right (resp. left) path is infinite. Moreover, since L is regular, Q has a finite cardinality and every right (resp. left) path is periodic. A path with a null pre-period is a *cycle*. The *strongly connected component* of p, denoted by $\mathrm{SCC}(p)$ is the set of states that belong to a cycle going through p.

The tuple $(p, z, z') \in T \times \Sigma^+ \times \Sigma^+$ is a *bi-path* if and only if $z \neq z'$, (p, z) and (p, z') are proper paths of $\mathcal{A}$ and $c(z) = c(z')$. It is sufficient that L be semi-geometrical for the condition $c(z) = c(z')$ to imply $p \cdot z = p \cdot z'$. Consequently, if (p, z, z') is a bi-path, then every point A of $\mathcal{F}(L)$ such that $\mathrm{State}(A) = p$ is the origin of a quasi-polygon (A, z, z'). The bi-path is said to be *empty* if the quasi-polygon is empty.

Let s and s' be two distinct words of Σ^+. Let us consider a bi-path $(p, ss', s's)$ such that (p, s) and (p, s') are proper and elementary paths of $\mathcal{A}$. Then, every point A of $\mathcal{F}(L)$ such that $\mathrm{State}(A) = p$ is the origin of a quasi-parallelogram $(A, ss', s's)$. Let us consider now the case where (p, s) and (p, s') are elementary cycles of $\mathcal{A}$. Then it holds $p \cdot s = p = p \cdot s'$ and the bi-path $(p, ss', s's)$ is called *a bi-cycle*. If $(p, ss', s's)$ is an empty bi-cycle then il is obvious that the condition $p \cdot s = p = p \cdot s'$ implies that the quasi-parallelogram $(A, ss', s's)$ is tiling. The following Lemma addresses the reciprocal.

Lemma 2. *Let p be a state of T and A be a point of $\mathcal{F}(L)$ such that* $\mathrm{State}(A) = p$. *Then, the two following conditions are equivalent:*

(1) *The bi-path* $(p, ss', s's)$ *is an empty bi-cycle.*
(2) *The empty quasi-parallelogram* $(A, ss', s's)$ *is a tiling one.*

Table 1. The transition function δ of $\mathcal{A}$

δ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	1	2	3	4	σ	σ	7	0	σ	σ	11	σ	σ	14	6
b	8	σ	12	σ	5	6	σ	σ	9	10	0	1	13	σ	σ

Let us consider the DFA defined by Table 1. Figure 3 represents the tiling induced by the maximal bi-cycle $(0, ss', s's)$, with $s = aaaabbaa$ and $s' = bbbb$. The quasi-parallelogram $(\mathcal{O}, ss', s's)$ is a puzzle made of three pieces, the holes of the 2-points $\mathcal{O}$, $\mathcal{O} + c(aa)$ and $\mathcal{O} + c(bbb)$.

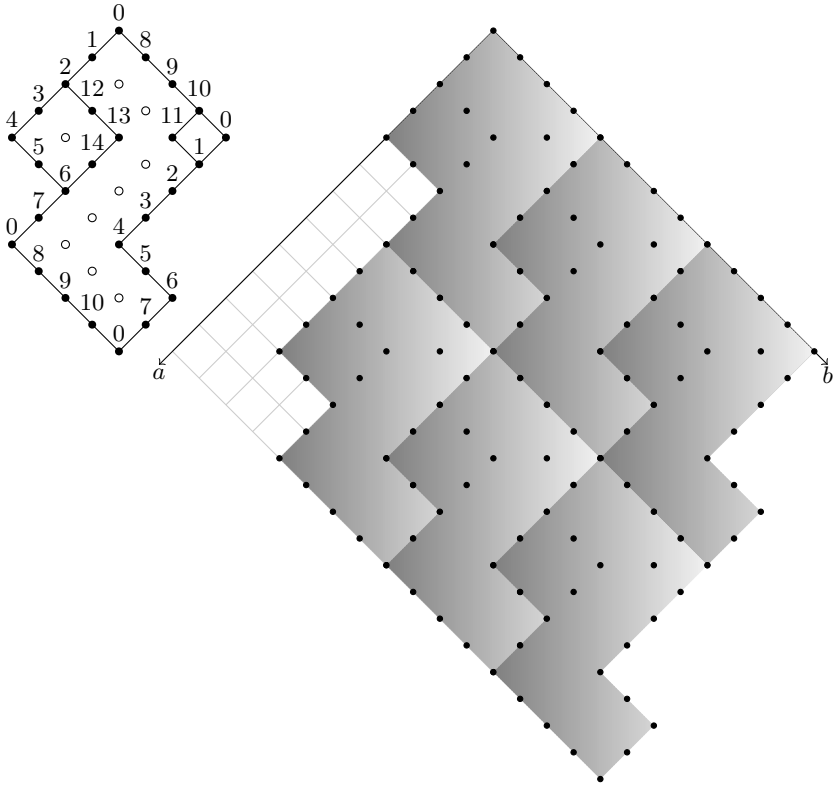


Fig. 3. Tiling induced by a maximal bi-cycle

If the bi-cycle $(p, ss', s's)$ is not empty, the following condition has to be checked for the quasi-parallelogram $(A, ss', s's)$ to be a tiling one.

Lemma 3. *Let* p *be a state of* T *and* A *be a point of* $\mathcal{F}(L)$ *such that* $\mathrm{State}(A) = p$. *Let us suppose that the bi-cycle* $(p, ss', s's)$ *is not necessarily empty. Then a necessary*

condition for the quasi-parallelogram $(A, ss', s's)$ *to be a tiling one is that the right and left paths of* p *have a null pre-period.*

The bi-cycle $(p, ss', s's)$ is said to be *maximal* if the cycles (p, s) and (p, s') respectively are the left path and the right path of p. If $(A, ss', s's)$ is tiling and if $\mathrm{State}(A) = p$, then the (s, s')-cone of A is the *cone* of p. Moreover, the *right* (resp. *left*) *trajectory* of P is associated with the right (resp. left) path of $\mathrm{State}(P)$. A point $P \in \mathcal{F}(L)$ is *a 2-point* if $P + c(a) \in \mathcal{F}(L)$ and $P + c(b) \in \mathcal{F}(L)$. The hole of a 2-point P of $\mathcal{F}(L)$ is the region delimited by the right trajectory of $(P + c(a))$ and the left trajectory of $(P + c(b))$. A hole is *convergent* if these trajectories intersect. The following lemma shows that the necessary condition of Lemma 3 is also a sufficient condition.

Lemma 4. *Let* p *be a state of* T *admitting a maximal bi-cycle* $(p, ss', s's)$ *and* A *be a point of* $\mathcal{F}(L)$ *such that* $\mathrm{State}(A) = p$. *Let* n_i *be the number of 2-points of the quasi-parallelogram* $(A, ss', s's)$. *Then this quasi-parallelogram is a puzzle made of* n_i *pieces, each of which is the hole of a 2-point.*

The following proposition is a corollary of Lemma 3 and Lemma 4. It shows a second difference with previous works dealing with tiling of the plane by polyominoes: here geometrical figures are built from regular languages and the tiling of the plane is actually deduced from a tiling of $\mathbb{N}^2 \times Q$.

Proposition 1. *Let* L *be a regular binary prolongable language. If* L *is geometrical, then for all state* p *of* T *such that* $(p, ss', s's)$ *is a maximal bi-cycle, there exists a tiling of the cone of the point* $A = \mathrm{Point}(p)$ *by the quasi-parallelogram* $(A, ss', s's)$.

5 State Sequences Associated with the Levels of a Tiling

We assume that L is a regular binary prolongable geometrical language. We consider the maximal bi-cycle $(p, ss', s's)$ and the tiling of the cone of $A = \mathrm{Point}(p)$ by $(A, ss', s's)$. In this section we compute the state sequence associated with the points where the horizontal line Δ_λ intersect the sides of the quasi-parallelograms of the tiling. This computation is illustrated by Figure 4.

Let $B = A + c(s)$, $D = A + c(s')$ and $C = A + c(ss')$. The *NE* (resp. *SE*) *side* of $(A, ss', s's)$ is the trajectory (A, s') (resp. (D, s)). The *West* (resp. *East*) *contour* is the trajectory (A, ss') (resp. $(A, s's)$). The *height* of a point R of $(A, ss', s's)$ is $\mathrm{height}(R) = \mathrm{level}(R) - \mathrm{level}(A)$. The point with height h on the West (resp. East) contour is denoted by W_h (resp. E_h). In a $p1$ type tiling every tile is obtained by translation from the reference tile. For all $(x, x') \in \mathbb{N}^2$, let us denote by $[x, x']$ the quasi-parallelogram $(M, ss', s's)$ obtained by the translation vector $xc(s) + x'c(s')$. Any quasi-parallelogram $[x, x']$ such that $x \geq 1$ admits two neighbours on its East contour: $[x - 1, x']$ and $[x, x' + 1]$. Any quasi-parallelogram $[0, x']$ admits one neighbour on its East contour: $[0, x' + 1]$. Let us set $\pi = |s|$, $\pi' = |s'|$ and $m = \pi + \pi'$.

Lemma 5. *Let* $[x, x']$ *be a quasi-parallelogram. If* $x \geq 1$, *for all* h, $0 \leq h < m$, *the point* E_h *of* $[x, x']$ *coincides with the point* $W_{h'}$ *of one of its East neighbours, with* $h' \equiv h + \pi \mod m$. *If* $x = 0$, *for all* h, $0 \leq h < \pi$, *the point* E_h *of* $[0, x']$ *is on the*

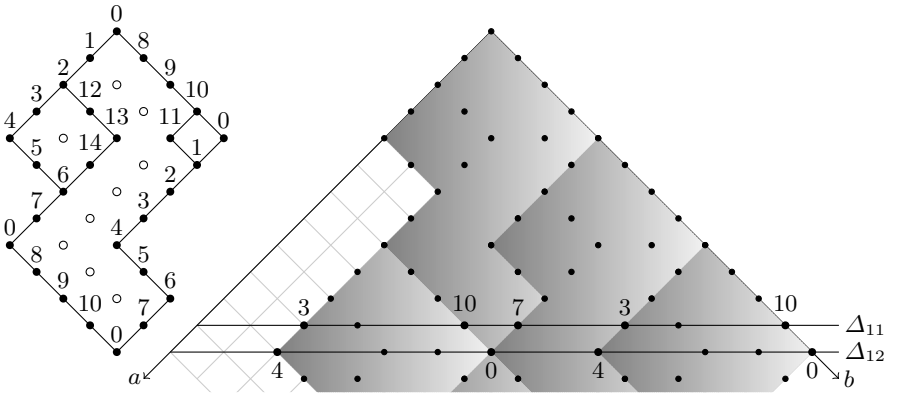


Fig. 4. Computation of the state sequence of a level

quasi-axis $Y_{s'}$; moreover, for all h, $\pi \leq h < m$, the point E_h of $[0, x']$ coincides with the point $W_{h'}$ of its SE neighbour, with $h' \equiv h + \pi \mod m$.

According to Lemma 5, for all h, $0 \leq h < m$, the computation of h' with respect to h only depends on π and π'; it can be performed on the quasi-parallelogram $(A, ss', s's)$ using the circular permutation φ defined by $\forall h$, $0 \leq h < m$, $\varphi(h) \equiv h + \pi \mod m$. Let us set $d = \gcd(\pi, m)$. According to Lemma 1, the permutation φ admits d orbits and, for all r, $0 \leq r < d$, it holds $O_r = (r + id)_{0 \leq i < m/d}$. For all k, $0 \leq k < m/d$, let us consider the sequence $O_{r,k} = (r + kd + id \mod m)_{0 \leq i < m/d}$.

Proposition 2. *Let $(p, ss', s's)$ be a maximal bi-cycle and $A = point(p)$. Let φ be the circular permutation associated with the quasi-parallelogram $(A, ss', s's)$, with $\pi = |s|$, $\pi' = |s'|$, $m = \pi + \pi'$ and $d = \gcd(\pi, m)$. Let S be the sequence of states of the path (p, ss'). Let Δ_λ be the horizontal line of level λ, with $\lambda \geq \mathrm{level}(A)$. Then the sequence $S_1 = (q_0, q_1, \ldots, q_{f-1})$ of the states associated with the intersection points of Δ_λ with the tiling by $(A, ss', s's)$ can be computed in the following way.*
(1) *Let us change the orbit O_0 by setting $O_0 = O_0 \setminus \{0\}$.*
(2) *The sequence $S_2 = (h_0, h_1, \ldots, h_{f-1})$ of the heights ot the intersection points is computed by making use of the permutation φ. The height h_0 is such that $\lambda \equiv h_0 \mod \pi$. Let us set $h_0 = r + kd$. Then S_2 is then made of the f first elements of the sequence obtained by repeating $O_{r,k}$ as many times as necessary.*
(3) *As a consequence it holds $S_1 = (S_{h_i})_{0 \leq i < f}$.*

6 Conclusion

Let L be a regular binary prolongable geometrical language. Our main result is that a necessary and sufficient condition for a point of the geometrical figure $\mathcal{F}(L)$ of L to be the origin of a tiling by a sub-figure of $\mathcal{F}(L)$ is that the associated state of the minimal DFA of $\mathrm{Pref}(L)$ admits both a cycle as left path and a cycle as right path. This property

of 2-dimensional geometrical languages is proved in the frame of the automaton-based characterization of geometrical languages, and it is expected to facilitate the study of d-dimensional geometrical languages.

References

1. Baruah, S.K., Rosier, L.E., Howell, R.R.: Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Systems 2(4), 301–324 (1990)
2. Beauquier, D., Nivat, M.: On translating one polyomino to tile the plane. Discrete & Computational Geometry 6, 575–592 (1991)
3. Blanpain, B., Champarnaud, J.-M., Dubernard, J.-P., Jeanne, H.: Geometrical languages. In: Martin Vide, C. (ed.) International Conference on Language Theory and Automata (LATA 2007), vol. 35/07, GRLMC Universitat Rovira I Virgili (2007)
4. Champarnaud, J.-M., Dubernard, J.-P., Jeanne, H.: Geometricity of binary regular languages. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 178–189. Springer, Heidelberg (2010)
5. Champarnaud, J.-M., Dubernard, J.-P., Jeanne, H.: An efficient algorithm to test whether a binary and prolongeable regular language is geometrical. Int. J. Found. Comput. Sci. 20(4), 763–774 (2009)
6. Eilenberg, S.: Automata, languages and machines, vol. B. Academic Press, New York (1976)
7. Geniet, D., Largeteau, G.: Wcet free time analysis of hard real-time systems on multiprocessors: A regular language-based model. Theor. Comput. Sci. 388(1-3), 26–52 (2007)
8. Golomb, S.W.: Polyominoes: Puzzles, patterns, problems, and packings. Princeton Academic Press, London (1996)
9. Kleene, S.: Representation of events in nerve nets and finite automata. Automata Studies, Ann. Math. Studies 34, 3–41 (1956)
10. Largeteau-Skapin, G., Geniet, D., Andres, E.: Discrete geometry applied in hard real-time systems validation. In: Andrès, É., Damiand, G., Lienhardt, P. (eds.) DGCI 2005. LNCS, vol. 3429, pp. 23–33. Springer, Heidelberg (2005)
11. Lungo, A.D.: Polyominoes defined by two vectors. Theor. Comput. Sci. 127(1), 187–198 (1994)
12. Myhill, J.: Finite automata and the representation of events. WADD TR-57-624, 112–137 (1957)
13. Nerode, A.: Linear automata transformation. In: Proceedings of AMS, vol. 9, pp. 541–544 (1958)
14. Parikh, R.: On context-free languages. J. ACM 13(4), 570–581 (1966)

COMPAS - A Computing Package for Synchronization

Krzysztof Chmiel[1] and Adam Roman[2]

[1] DreamLab Onet.pl sp. z o.o., Gabrieli Zapolskiej 44, 30-126 Kraków, Poland
[2] Institute of Computer Science, Jagiellonian University in Cracow, Poland
tikan@autocom.pl, roman@ii.uj.edu.pl

Abstract. In this paper we describe COMPAS - the open-source computing package, dedicated to the computations on synchronizing automata. COMPAS design is based on a generic programming paradigm. This makes the package very powerful because of its flexibility and extensibility. The paper describes shortly the package architecture and its main algorithms and some examples of use. COMPAS allows to easily operate on synchronizing automata, verifying new synchronizing algorithms etc. To the best of our knowledge, this is the first such flexible, extensible and open-source package for synchronization.

1 Introduction and Motivation

A finite automaton $\mathcal{A}$ is synchronizing, if there exists a state q and some word w that takes all automaton states to q. The word w is called a synchronizing word for $\mathcal{A}$. The shortest such w is called a minimal synchronizing word. In the past few years synchronization of finite automata has attracted attention of many researchers. The main reason is the famous Černý Conjecture - an open problem since 1964. Černý Conjecture claims that if an n-state automaton $\mathcal{A}$ is synchronizing, then its minimal synchronizing word is no longer than $(n-1)^2$. On the other hand, synchronization theory has many practical applications, such as simple error recovery in finite automata or leader identification in processor networks [13]. Since 60s till 90s synchronizing automata were considered as a useful tool for testing of reactive systems (circuits, protocols) [27]. In the 80s synchronizing theory was used in robotics - an interesting example with so-called part orienters is given in [2].

Some results in synchronization theory required a numerical computations or could be easily obtained with computer use. For example, in [11] Kari gave a counterexample to the so-called Černý-Pin Conjecture. This result could be obtained by checking directly some property of the minimal synchronizing word for all six state synchronizing automata. In [26] Trahtman did computations for checking the lengths of the minimal synchronizing words for automata with 2,3,...,10 states. He discovered some interesting trend for the length of the longest minimal synchronizing word shorter that $(n-1)^2$ among all n-state automata, $n = 2, 3, ..., 10$. In [20] the first automaton over 3-element alphabet, obtaining the

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 79–86, 2011.

Černý upper bound, was presented. The automaton was found by the computer. It is known that the Černý Conjecture holds true for all automata with less than $n = 11$ states over binary alphabet. Again, for $n = 7, 8, 9, 10$ the result was obtained by the numerical computations. With powerful computers and efficient software, one could examine some bigger automata, or automata with alphabet size greater than 2, and check if the conjecture still holds. Recently, motivated by the work of Higgins [10], some authors began the research on the random automata synchronization (cf. [27,24]). Two main questions in this new line of research are: 1) what alphabet size s guarantees that with a high probability the random automaton over s-letter alphabet is synchronizing and 2) what alphabet size t guarantees that a random automaton over t-letter alphabet obey the Černý Conjecture? By 'high probability' we understand that the probability tends to 1 when the number of states goes to infinity. The exact lower bound for the alphabet size is still not known. The numerical experiments could shed some light on it. The computing package can be very helpful here.

From the practical point of view, the most important task in synchronization is the following: given an automaton, find the synchronizing word as short as possible. It is known that the problem of finding the minimal synchronizing word is both NP-hard and coNP-hard and the problem of finding a synchronizing word of a given length is NP-complete (see [23,8]). Therefore, assuming $P \neq NP$, for bigger automata we can only rely on heuristic, polynomial algorithms. There are some well-known algorithms, such as Natarajan's or Eppstein's one [17,8] or some newer [21,26]. Having a framework which could allow to generate all synchronizing automata of a given number of states, one can compare the algorithms effectiveness. An example of some partial comparison can be found in [21] and [26].

Each above problem requires a different algorithm to use. Of course it is possible to write a completely new program for each new problem, but a much wiser solution is to create a framework, which could allow to solve these problems in a fast, easy and efficient way. There are many packages dealing with automata, semigroups, grammars and regular expressions (cf. [5,9,15]), but none of them is dedicated directly to the synchronizing issues. The only existing such package is TESTAS [25], but its functionality is restricted to some predefined algorithms. In this paper we present COMPAS - a new **com**puting **pa**ckage for **s**ynchronization. It is a free, open-source package written in C++. COMPAS allows not only using its embedded functions in a batch mode (like in Matlab), but also writing new procedures, scripts or even standalone compiled programs, which use COMPAS as a library. We also used the Python-C++ binding to allow a user writing scripts in Python, which is a very 'readable' language.

2 Synchronizing Automata and Synchronizing Algorithms

A *deterministic finite automaton* (DFA) is a triple $\mathcal{A} = (Q, A, \delta)$, where Q is a nonempty, finite set of states, A is a finite alphabet and $\delta : Q \times A \to Q$ is the transition function, called also the automaton action. By A^* we denote the free

monoid over A, consisting of all finite words over A. By ε we denote an empty word of length 0. It is convenient to extend the δ function on all subsets in the usual way: for $P \subset Q$ we define: $\delta(P, \varepsilon) = P$, $\delta(P, a) = \cup_{p \in P}\{\delta(p, a)\}$. We say that $w \in A^*$ *synchronizes* $\mathcal{A} = (Q, A, \delta)$ if $|\delta(Q, w)| = 1$. If such a word exists, $\mathcal{A}$ is called a *synchronizing automaton*. Notice, that if $w \in A^*$ is a synchronizing word for $\mathcal{A}$, then for all $u, v \in A^*$ uwv also synchronizes $\mathcal{A}$. A natural question arises: what is the *shortest* synchronizing word for a given DFA? This question is also of a practical nature - in the real applications, as described in Section 1, longer synchronizing word means usually higher cost of some action (time, money etc.).

Let $S \subset A^*$ be the set of all synchronizing words for some synchronizing DFA $\mathcal{A} = (Q, A, \delta)$. A word $s \in S$, such that $\forall t \in S\ |s| \leq |t|$ is called a *minimal synchronizing word* for $\mathcal{A}$. Notice, that it is possible for a synchronizing DFA to have more than one minimal synchronizing word. From the theoretical point of view it is interesting to ask about the relationship between the number of states of $\mathcal{A}$ and the length of the minimal synchronizing word for $\mathcal{A}$. Let Π_k denote the set of all k-state synchronizing DFA (which are all different up to the labeling of Q and of A) and let $\mathcal{A} = (Q, A, \delta) \in \Pi_k$. Define $n(\mathcal{A}) = \min\{n \in \mathbb{N} |\ \exists w \in A^*,\ |w| = n :\ |\delta(Q, w)| = 1\}$ and $n(k) = \max_{\mathcal{A} \in \Pi_k}\{n(\mathcal{A})\}$. In order to find a lower bound for $n(k)$, Černý [6] introduced a family $\mathcal{C}$ of automata, such that $|\mathcal{C} \cap \Pi_k| = 1\ \forall k \geq 2$. These automata are called the Černý automata. We denote by $\mathcal{C}_k$ a k-state Černý automaton. $\mathcal{C}_k = (\{0, 1, ..., k-1\}, \{a, b\}, \delta)$, where

$$\delta(i, x) = \begin{cases} i + 1 \bmod k & \text{if } x = a \\ i & \text{if } x = b \wedge i < k-1 \\ 0 & \text{if } x = b \wedge i = k-1 \end{cases}$$

Černý proved that the minimal synchronizing word for $\mathcal{C}_k$ equals $(k-1)^2$. In 1971 he stated the famous Černy Conjecture:

Conjecture 1 (Černý). Let $\mathcal{A} \in \Pi_k$ and let w be the minimal synchronizing word for $\mathcal{A}$. Then $|w| \leq (k-1)^2$.

The conjecture was shown to be true for certain classes of automata (cf. [7,12]), but in general case it is an open problem. The best known upper bound for $n(k)$ is $\frac{k^3-k}{6}$ and this is a result of Pin [19]. Another, independent proof can be found in [14]. The above results can be recapitulated in the following inequalities:

$$(k-1)^2 \leq n(k) \leq \frac{k^3 - k}{6}.$$

Now, we will shortly describe an algorithmic approach to the synchronization. There are three main algorithmic problems concerning the synchronizing property:

(P1) Given a DFA $\mathcal{A}$, check if $\mathcal{A}$ is synchronizing.
(P2) Given a synchronizing DFA $\mathcal{A}$, find a synchronizing word for $\mathcal{A}$.
(P3) Given a synchronizing DFA $\mathcal{A}$, find a minimal synchronizing word for $\mathcal{A}$.

(P1) can be solved in $O(|A||Q|^2)$, by checking the connectiveness of a so-called pair-automaton for $\mathcal{A}$. For (P2) we can use a modification of (P1) algorithm. This is a well-known Eppstein algorithm [8] with $O(|Q|^3 + |A||Q|^2)$ time complexity. Decision version of (P3) is NP-hard, so assuming $P \neq NP$ we can use only an exponential algorithm with $O(|A|2^{|Q|})$ time complexity.

3 COMPAS Architecture and Functionality

COMPAS consists of three main parts: 1) a library (further called also a framework) written in C++, which implements automata of different types and algorithms working on them; 2) a module that allows using the library at the script level (in Python); 3) GUI, in which one can write scripts in Python (a'la Matlab).

Conceptual Design. Before we start implementing any bigger library, we have to answer some fundamental questions: what do we really want to achieve? What are the most important features of the library? In practice it is hardly possible to implement a program that brings together good flexibility, scalability and usability. Of course it is worth a try, but usually we have to compromise. Likewise, in COMPAS we stress on some (in our opinion, more important) aspects of its functionality. The aims, graded in decreasing essentiality order, are the following:

- **Genericity, flexibility, scalability.** A generic library gives to the programmer the freedom in choosing data structures. The implementation of different objects is not predetermined. This also makes the library algorithms capable to solve effectively problems of different character (e.g., operations on deterministic and non-deterministic automata) and of different size (e.g. operations on automata with total transition function and on automata with partially defined transition function).
- **Efficiency.** We want COMPAS algorithms to be as effective as the algorithms written 'from scratch' in C or C++ and dedicated directly to solve some specific problems.
- **Usability.** The library must be easy to use. The programmer, after skimming over the existing code, should be able to create his own automata and use the basic algorithms. We don't assume a user to be an expert in C++. A user with some basic knowledge on C++ should be able to use the majority of package functions. Achieving the burst performance of COMPAS and implementing effective algorithms requires a good or very good C++ knowledge.

We chose C++ for the COMPAS implementation. There were few reasons for that, but maybe the most important one is the ability to write clear, elegant, high-level code, keeping at the same time the high effectiveness (in some cases, comparable to the effectiveness of a low-level code). When it is needed to optimize some parts of the algorithms, one can go to a low level and write some

code in C. This is a very useful feature of C++. Due to a popularity of C++, there are many high quality libraries for it. One of them is the *boost* library [4], intensively utilized in our package. Boost extends the C++ capabilities, being somehow supplementary to the standard library STL. Last but not least, in C++ we can use the template-based static polymorphism and in fact we make use of that.

Static Polymorphism. In short, polymorphism is a language property, that allows using objects of different types with the common, uniform interface. This notion is independent of programming paradigm, but it is commonly found in object-oriented languages. The classical way to implement polymorphism in OO languages is to use inheritance in connection with virtual functions. This technique can be used in C++, but in COMPAS our approach rely on a so-called static polymorphism. It is based on the class and function templates and is realized during the compilation phase. The main advantage of this approach is increased performance - there is no waste of time for realizing virtual calls (while the time spent on a single virtual call is marginal, it becomes an important factor in time complexity for the execution of short and frequently used function).

Concepts. Concept is a key notion in static polymorphism - it is a set of requirements imposed on the type being a parameter of a given template. A concept is parallel to the interface in classical OO programming. Simple examples of concepts (taken from the standard library) are: Assignable (a type with assign operator), CopyConstructible (types that must be able to be constructed from another member of the type), WeakComparable (a type with $<$ operator defining the partial order). In a real life, concepts (like **Automaton** concept from our library) are much more complicated. In addition to type requirements they can impose requirements for constructors, associated types or just certain typedef declarations nested in the class. It is surprising that concepts, as for so important notion, are hardly supported by contemporary C++ compilers. One can say that they are present in documentation and they are not specified in code. Type checking takes place at the stage of creating the template instance, when a specified property (method, field) of a given type is used. Thereby, in case of some incompatibility, error messages are often illegible and illusorily not related to the real cause of the problem. There are external libraries that deal with this (one of them is available in the boost library).

Framework Structure. In general, the framework consists of two parts: the first one is a set of templates that implement finite automata and the basic operations on them. The second one is a set of algorithms working on automata. This part includes several synchronizing algorithms. The library is, in some sense, two-level generic. First, generic is the automata implementation itself: a user can configure many parameters that have influence on properties, behavior and efficiency. Next, the algorithms itself are generic - they don't depend on the way that a given automaton is implemented.

COMPAS Algorithms. There are four basic algorithms for automata in COMPAS: first two check if an automaton is deterministic and total (that is, if δ is a total function), third one reverses the automaton trasitions (note, that the reversed automaton type is usually more general than the type of an initial automaton), and the last one copies an automaton with the possibility of type change; this function accepts two template parameters: original automaton type and output automaton type. All these algorithms are implemented in a very simple way and run in a linear time. There are several functions implemented in COMPAS, responsible for the fast creation of some special synchronizing automata that are important from a theoretical point of view [6,11,16,3]. It is also possible to generate a random automaton, an identity automaton and some other types of automata. Synchronization is the most important issue for COMPAS. Till now, four different synchronizing algorithms are implemented: expomential algorithm, Eppstein's greedy algorithm, cycle algorithm (a modification of Eppstein alg.) and genetic algorithm [22].

COMPAS contains also some other useful algorithms. Here we describe only some of them. These are algorithms which allow: to build a power automaton; to build a product automaton; to check if an automaton is Eulerian; to check if two automata are isomorphic up to the state labeling and alphabet; to check if two automata are equal. One of the useful functions is *next_automaton*, which, for a given automaton $\mathcal{A}$, returns the next automaton in a specific order $\prec$. To define $\prec$, let us introduce the family of mappings $S_{n,m} : \mathbf{A}_{n,m} \to \{1, ..., n\}^{n \times m}$, parameterized by n and m, where $\mathbf{A}_{n,m}$ denotes the family of n-state automata over m-letter alphabet. For a given automaton $\mathcal{A} = (\{q_1, ..., q_n\}, \{a_1, ..., a_m\}, \delta)$ we put $S_{n,m}(A) = (\delta(q_1, a_1), ..., \delta(q_1, a_n), ..., \delta(q_n, a_m))$. Let $<_l$ be the lexicographic order on $\{1, ..., n\}^{n \times m}$. Formally, $\mathcal{A}_1 \prec \mathcal{A}_2 \Leftrightarrow S_{n,m}(\mathcal{A}_1) <_l S_{n,m}(\mathcal{A}_2)$.

Acknowledgements and Future Work

COMPAS can be downloaded from http://www.assembla.com/spaces/compas. On http://www.ii.uj.edu.pl/∼roman/publications.html one can find the examples of experiments on synchronizing automata written with COMPAS (for example, finding the counterexample to the Černý-Pin Conjecture, finding automata reaching the upper bound $(n-1)^2$ from the Černý Conjecture and so on. Due to the page limit restrictions we cannot present them in the paper.

We would like to stress that the COMPAS package is a young project and it is still being developed. In the nearest future we plan to provide:

- a detailed English documentation,
- some standard I/O formats for presenting finite automata, like the Berkeley KISS2 format,
- a completely new, much more advanced GUI (the work has already been started),
- algorithms for presenting automata graphically in an elegant way,
- a new synchronizing algorithm, currently being developed by one of the authors.

We will also improve the effectiveness of some already implemented functions, like generating all non-isomorphic n-state automata using a recently described method from [1]. Notice also that the main idea of COMPAS is not to provide some number of predefined synchronizing algorithms, but rather to provide to the user the library, which allows to implement her/his own scripts, synchronizing algorithms etc.

The authors would like to thank Stefan Chrobot, Adam Radzimowski and Andrzej Kukier, who had their significant contribution in the initial phase of the project and in application testing. We would also like to thank the anonymous referees for their many valuable comments and ideas.

References

1. Almeida, M., Moreira, N., Reis, R.: Enumeration and generation with a string automata representation. Theor. Comp. Sci. 387(2), 93–102 (2007)
2. Ananichev, D.S., Volkov, M.: Synchronizing monotonic automata. Theor. Comp. Sci. 327(3), 225–239 (2004)
3. Ananichev, D.S., Volkov, M., Zaks, Y.I.: Synchronizing automata with a letter of deficiency 2. Theor. Comp. Sci. 376(1-2), 30–41 (2007)
4. boost library, `http://www.boost.org`
5. Camparnaud, J.M., Hansel, G.: Automate, a computing package for automata and finite semigroups. J. Symb. Comput. 12, 197–220 (1991)
6. Černý, J.: Poznámka k komogénnym experimentom s konečnými automatmi. Mat. fyz. čas. SAV 14, 208–215 (1964)
7. Dubuc, L.: Sur les automates circulaires et la conjecture de Černý. RAIRO Inf. Theor. Appl. 32, 21–34 (1998)
8. Eppstein, D.: Reset sequences for monotonic automata. SIAM J. Comp. 19, 500–510 (1990)
9. Froidure, V., Pin, J.E.: Algorithms for computing finite semigroups. Foundations of Comp. Math., 112–126 (1997)
10. Higgins, P.M.: The range order of a product of i transformations from a finite full transformation semigroup. Semigroup Forum 37, 31–36 (1988)
11. Kari, J.: A Counter Example to a Conjecture Concerning Synchronizing Words in Finite Automata. Bull. EATCS 73, 146 (2001)
12. Kari, J.: Synchronizing finite automata on Eulerian digraphs. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, pp. 432–438. Springer, Heidelberg (2001)
13. Kari, J.: Synchronization and Stability of Finite Automata. J. Universal Comp. Sci. 8(2), 270–277
14. Klyachko, A.A., Rystsov, I.C., Spivak, M.A.: An extremal combinatorial problem associated with the bound of the length of a synchronizing word in an automaton. Cybernetics 23, 165–171 (1987)
15. Lombardy, S., Régis-Gianas, Y., Sakarovitch, J.: Introducing VAUCANSON. Theor. Comp. Sci. 328(1-2), 77–96 (2004)
16. Martjugin, P.V.: A series of slowly synchronizing automata with a zero state over a small alphabet. Inform. and Comput. 206(9-10), 1197–1203 (2008)
17. Natarajan, B.K.: An algorithmic approach to the automated design of parts orienters. In: 27th Annual Symposium on Foundations of Computer Science, pp. 132–142. IEEE, Los Alamitos (1986)

18. Pin, J.-E.: Sur un cas pariculier de la conjecture de Černý. In: Ausiello, G., Böhm, C. (eds.) ICALP 1978. LNCS, vol. 62, pp. 345–352. Springer, Heidelberg (1978)
19. Pin, J.-E.: On two combinatorial problems arising from automata theory. Ann. of Discr. Math. 17, 535–548 (1983)
20. Roman, A.: A Note on Černý Conjecture for Automata over 3-Letter Alphabet. J. Aut. Lang. and Comb. 13(2), 141–143 (2008)
21. Roman, A.: Synchronizing finite automata with short reset words. Appl. Math. Comp. 209(1), 125–136 (2009)
22. Roman, A.: Genetic Algorithm for Synchronization. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 684–695. Springer, Heidelberg (2009)
23. Samotij, W.: A note on the complexity of the problem of finding shortest synchronizing words. In: Proc. AutoMathA 2007, Automata: from Mathematics to Applications, Univ. Palermo, CD (2007)
24. Skvortsov, E., Zaks, Y.: Synchronizing Random Automata. In: AutoMathA 2009 Conference, to be published in Discr. Math. and Theor. Comp. Sci. (2009)
25. Trahtman, A.N.: A Package TESTAS for Checking Some Kinds of Testability. In: Champarnaud, J.-M., Maurel, D. (eds.) CIAA 2002. LNCS, vol. 2608, pp. 228–232. Springer, Heidelberg (2003)
26. Trahtman, A.N.: An efficient algorithm finds noticeable trends and examples concerning the Černý conjecture. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 789–800. Springer, Heidelberg (2006)
27. Volkov, M.: Synchronizing automata and the Road Coloring Theorem. In: A Tutorial on a Satellite Workshop to CSR 2008 "Workshop on Algebra, Combinatorics and Complexity", Moscow 2008 (2008)

From Sequential Extended Regular Expressions to NFA with Symbolic Labels*

Alessandro Cimatti, Sergio Mover, Marco Roveri, and Stefano Tonetta

Fondazione Bruno Kessler - IRST

Abstract. Practical property specification languages such as the IEEE standard PSL use at their core Sequential Extended Regular Expressions (SERE). In order to enable the reuse of traditional verification techniques, it is necessary to translate SEREs into automata. SERE are regular expressions built over alphabets resulting from the state variables of the design under analysis. Thus, a traditional approach to generate the automaton would suffer from the fact that the size of the alphabet is exponential in the number of symbols in the design.

In this work, we tackle this problem by proposing non-deterministic finite automata with symbolic representation of transitions labels, by way of propositional formulas, while states and transitions are explicitly represented. We provide a symbolic version of the algorithms for all the major operations over non-deterministic finite automata. The approach has been implemented in the AUTLIB library, with Binary Decision Diagrams (BDD) used to represent transition labels.

We carried out a thorough experimental evaluation over a set of realistic benchmarks, comparing our library against MONA (which uses deterministic finite automata with BDD-based symbolic transitions), and against GRAZ (which features non-deterministic finite automata with a DNF-based representation of the labels). Experimental results over a realistic set of benchmarks show that both features of AUTLIB (the ability to deal with non-determinism, and a BDD-based treatment of labels) are fundamental to achieve acceptable performance.

1 Introduction

Property specification languages (e.g. the IEEE standard PSL [1] and SVA [19]) are increasingly used to represent requirements of hardware components. Such languages extend the power of temporal operators [14] by featuring at their core an extended form of regular expressions, called SERE. In order to generalize well-established model checking techniques [9] from traditional temporal logics to such languages, recent approaches [8] require the ability to generate a finite automaton accepting the language of a given SERE.

There exist well-known solutions to the automata construction from regular expression (cfr., e.g., [11,6,4,20]). These approaches are in principle valid in the context of SEREs, which also represent regular languages. However, they are inefficient in the context of SEREs mainly for the following reasons. First, the alphabet of a SERE over a set of atomic propositions $\mathcal{P}$ is $\Sigma_{\mathcal{P}} = 2^{\mathcal{P}}$, i.e. the powerset of the set of atomic propositions. This means that the size of the alphabet grows exponentially in the number of

* S. Tonetta is supported by the Provincia Autonoma di Trento (project ANACONDA). The other authors are supported by EU grant FP7-2007-IST-1-217069 COCONUT.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 87–94, 2011.

atomic propositions. In real PSL formulas, the number of atomic propositions can be large, thus inducing a huge alphabet. Second, SEREs are concise using Boolean formulas over atomic propositions as atomic regular expressions. Boolean formulas represent a set of letters of alphabet, a classical automata construction would force an explicit enumeration of the letters (i.e. the models of a formula), which is exponential.

We tackle this problem by proposing non-deterministic finite automata with a symbolic representation of transitions labels (NFASL). The idea is to represent explicitly states and transitions, and to represent the set of all transitions between two states with just one transition labeled with a Boolean formula. This representation is more concise, since multiple transitions can be collapsed together. Moreover, it avoids an explicit enumerations of the alphabet letters, at the price of applying symbolic transformations when combining transition labels. To this extent, we provide a symbolic version of the algorithms for all the major operations over NFASL, and a procedure to map a SERE into an NFASL. The approach has been implemented in the AUTLIB library, using Binary Decision Diagrams (BDDs) to represent transition labels.

We carried out a thorough experimental evaluation over a set of realistic benchmarks, comparing our AUTLIB library against MONA and GRAZ. MONA is a well known and optimized BDD-based procedure that uses deterministic finite automata with a symbolic representation of the transition relation. GRAZ is a library that features non-deterministic finite automata with a semi-symbolic representation of transition labels, based on Disjunctive Normal Form (DNF). Experimental results over a realistic set of benchmarks show substantial advantages over either competitor, and substantiate the claim that both features of AUTLIB (the ability to deal with non-determinism, and a fully symbolic treatment of labels) are fundamental to achieve acceptable performance.

The paper is structured as follows. In Sec. 2 we present some background on SERE. In Sec. 3 we formalize the notion of NFASLs. In Sec. 4 we compare our approach with related work, and in Sec. 5 we experimentally evaluate the AUTLIB library. In Sec. 6 we draw some conclusions and outline directions for future research.

2 Regular Expressions for Property Specification

PSL is an IEEE-standard language [1] for the specification of hardware requirements, based on a combination of Linear Temporal Logic [14] with SERE, a variant of classic regular expressions [11]. A key difference of SEREs with respect to classic regular expressions is that a letter in the alphabet of a SERE is a truth assignment to a set of atomic propositions, since SEREs are defined over Boolean formulas. In this section we formally define syntax and semantics of SEREs.

Notation. We consider regular languages parameterized by a set of atomic propositions $\mathcal{P}$. The alphabet of such languages is given by the set $\Sigma_{\mathcal{P}} = 2^{\mathcal{P}}$ of truth assignments to the propositions of $\mathcal{P}$. We will use $\mathcal{B}_{\mathcal{P}}$ to denote the set of Boolean formulas (obtained applying conjunction $\wedge$, disjunction $\vee$ and negation $\neg$) over the elements of $\mathcal{P}$. We use ℓ,ℓ', ℓ_1,...,ℓ_n to refer to a letter in the alphabet $\Sigma_{\mathcal{P}}$. A finite word is a finite sequence of letters. We use v, w, w_1, w_2 to denote finite words and $\Sigma^*_{\mathcal{P}}$ to denote the set of all words in $\Sigma_{\mathcal{P}}$. Given $v = \ell_0, \ell_1, \ldots, \ell_n \in \Sigma^*_{\mathcal{P}}$ and $w = \ell'_0, \ell'_1, \ldots, \ell'_n \in \Sigma^*_{\mathcal{P}}$,

$vw = \ell_0, \ell_1, \ldots, \ell_n, \ell'_0, \ell'_1, \ldots, \ell'_n$ is the concatenation of words v and w. w^0 denotes the first letter of w.

Syntax and Semantics. The syntax of SEREs is defined as follows:

Definition 1 (SEREs syntax). *An atomic expression is either a Boolean formula $\phi \in \mathcal{B}_\mathcal{P}$ or the empty word denoted with ϵ.* SEREs *are obtained by applying recursively the following operators to the atomic expressions:*

- *if r_1, r_2 are* SEREs*, then r_1 **;** r_2, r_1 **:** r_2, r_1 **|** r_2, r_1 **&** r_2 and r_1 **&&** r_2 are* SEREs*;*
- *if r is a* SERE*, then r**[*]** and r**[+]** are* SEREs*.*

Boolean formulas are interpreted over letters in $\Sigma_\mathcal{P}$: a propositional atom p is true in ℓ iff $p \in \ell$, false otherwise; Boolean connectives are interpreted in the standard way. If ℓ is a letter and b a Boolean formula, we denote with $\ell \models_\mathcal{B} b$ by the fact that ℓ is a model of b.

Definition 2 (SERE semantics). *Let w be a finite word over $\Sigma_\mathcal{P}$, ϕ a Boolean formula, r, $r1$, $r2$* SEREs*, then the satisfaction relation $w \models r$ is defined as follows:*

- $w \models \epsilon$ iff $|w| = 0$
- $w \models \phi$ iff $|w| = 1$ and $w^0 \models_\mathcal{B} \phi$
- $w \models r_1$ **;** r_2 iff $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_1$, $w_2 \models r_2$
- $w \models r_1$ **:** r_2 iff $\exists w_1, w_2, \ell$ s.t. $w = w_1 \ell w_2$, $w_1 \ell \models r_1$, $\ell w_2 \models r_2$
- $w \models r_1$ **|** r_2 iff $w \models r_1$ or $w \models r_2$
- $w \models r_1$ **&** r_2 iff *either* $w \models r_1$ and $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_2$, or $w \models r_2$ and $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_1$
- $w \models r_1$ **&&** r_2 iff $w \models r_1$ and $w \models r_2$
- $w \models r$**[*]** iff $|w| = 0$ or $\exists w_1, w_2$ s.t. $|w_1| \neq 0$, $w = w_1 w_2$, $w_1 \models r$, $w_2 \models r$**[*]**
- $w \models r$**[+]** iff $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r$, $w_2 \models r$**[*]**

Definition 3 (Language of SEREs). *The language of a* SERE *r is the set $\mathcal{L}(r) := \{w \in {\Sigma_\mathcal{P}}^* \mid w \models r\}$.*

Example 1. The SERE $\{start$**;**$\{busy\}$**[*];**$end\}$**&&**$\{\{\neg abort\}$**[*]**$\}$ over $\mathcal{P} = \{start, busy, end, abort\}$ may represent the sequences of a potential hardware procedure that lasts for an uncertain number of cycles while never aborted. The SERE $\{req$**;**$\{\{read$ **;**$\{\neg cancel\_r \wedge \neg done\}$**[*]**$\}$ **|** $\{write$**;**$\{\neg cancel\_w \wedge \neg done\}$**[*]**$\}\}$**;**$done\}$ over $\mathcal{P} = \{req, read, write, cancel\_r, cancel\_w, done\}$ may represent the sequences of a request of read or write which is accomplished without being canceled.

3 Non-deterministic Finite Automata with Symbolic Labels

Definition 4 (NFASL). *A* Non-deterministic Finite-state Automaton with Symbolic Labels *(NFASL) is a tuple $A = \langle \mathcal{P}, Q, q^0, \rho, F \rangle$, where $\mathcal{P}$ is the set of atomic propositions, Q is a finite set of states, $q^0 \in Q$ is the initial state, $\rho : Q \to 2^{\mathcal{B}_\mathcal{P} \times Q}$ is the symbolic transition function and $F \subseteq Q$ is the set of final states.*

By definition, an NFASL can move from a state q non-deterministically choosing a pair of label-state. The definition of transition function differs from the classic one,

where a move from state q is determined by a single letter of the alphabet, namely $\rho_C : Q \times \Sigma_{\mathcal{P}} \rightarrow 2^Q$. Given, ρ, we can define ρ_C as $\rho_C(q, l) := \{q' \mid \langle \phi, q' \rangle \in \rho(q)$ for some ϕ s.t. $l \models \phi\}$. A tuple $\langle q, \phi, q' \rangle$, where $\langle \phi, q' \rangle \in \rho(q)$, is called a *symbolic transition*. A symbolic transition $\langle q, \phi, q' \rangle$ is said *feasible* iff ϕ is satisfiable.

Definition 5 (NFASL language). *An NFASL $A = \langle \mathcal{P}, Q, q^0, \rho, F \rangle$ accepts a word $l_1, \ldots, l_n \in \Sigma^*_{\mathcal{P}}$ iff there exists a sequence of states $\pi = q_0, q_1, \ldots, q_n$ such that $q_0 = q^0$, $q_n \in F$, and, for all i, $1 \leq i \leq n$, there exists $\phi_i \in \mathcal{B}_{\mathcal{P}}$ such that $\langle \phi_i, q_i \rangle \in \rho(q_{i-1})$ and $l_i \models \phi_i$. The set $L(A) \subseteq \Sigma^*_{\mathcal{P}}$ of words accepted by A is called* language *of A.*

We convert a SERE into an equivalent NFASL using a variant of the Berry-Sethi construction (cfr. [20]). The algorithm builds the automaton in a bottom-up fashion, recursively applying automata operations and exploiting the symbolic representation of labels. For example, to intersect two automata we build the cross product of their states and for each pair of transitions we take the conjunction of the labels. In the algorithm we never add transitions with unsatisfiable labels and states that are not reachable from the initial state. The construction matches the complexity of the standard algorithms, in that it builds an automaton with a linear number of states if the SERE does not contain any intersection operator, while it is in general exponential.

We also implemented the determinization and state reduction operations. The determinization operation is a modified version of the subset construction algorithm. When processing a state we do not compute for every letter the set of states where the automaton can move, since this requires to enumerate all the possible truth assignments for every label. Instead, we create a transition for every combination of labels: these transitions are deterministic, since all the combinations of labels do not have any common assignment. We perform the reduction of the state space of a NFASL using the quotient graph with regard to the bi-simulation relation. This is a standard technique (cfr., e.g., [12]), but we adapt the definition of simulation to the symbolic labels.

A detailed description of automata operations is reported in an extended version of the paper available at `http://es.fbk.eu/people/mover/paper/CIAA10/`.

4 Related Work

Several works focus on the construction of automata from regular expressions. Most of them (e.g. [13,16,5]) use classic automata representations, where states and labels are represented explicitly. Other implementations (e.g., AUTOMATA[1] and LIBFA[2]) have a partially-symbolic representation of labels (e.g., intervals of letters). However, these approaches are inefficient when considering Boolean formulas as atomic expressions.

Symbolic representations of finite state automata have been investigated in several works. As in our approach, MONA [10] uses an explicit representation for states, but a symbolic representation for the entire transition relation. The symbolic representation is achieved using a variant of BDDs, called shared multi-terminal BDDs (SMBDDs).

[1] `http://www.brics.dk/automaton/`
[2] `http://augeas.net/libfa/`

Roots and leaves in a SMBDD are states of the automaton, while the internal nodes are atomic propositions. A transition is represented with a path from a root to a leaf. Unlike our approach, MONA cannot represent NFAs: given a state and a letter, there is a unique leaf node. Moreover, MONA does not implement regular languages operations such as concatenation and Kleene closure. STRANGER [21] is a library developed in the context of static strings analysis for Web applications. It is implemented on top of MONA, and extends it with more operations on automata.

Other approaches represent automata states and transitions explicitly and use a symbolic representation of labels, as in our case. The GRAZ library [15] represents NFAs where transitions are labeled with DNF formulas. Pairs of labels are combined by multiplying all the disjuncts of the first label with the disjuncts of the second label. This library was previously used in the NUSMV [7] model checker to manage the construction of automata from SEREs [8].Also in FSA [17] a predicate is used as label for a transition. The library uses Prolog and not BDDs to represent a predicate and to check its satisfiability. FSA describes the algorithms that we use to perform intersection and determinization. In [2] the authors give an efficient implementation of minimization for NFAs with large alphabets. The representation of NFA is explicit for states, and symbolic (BDD-based) for labels. This work does not take into account the construction of automata from regular expressions. Also REX [18] uses an approach similar in spirit to ours, where states are explicit and labels are symbolic. The key difference is that reasoning on symbolic labels is done using a Satisfiability Modulo Theory (SMT) solver instead of BDDs. As ours, the approach aims at dealing with extended regular expressions. However not all SEREs operations are covered (e.g. the fusion operator is missing).

In the context of circuit synthesis for PSL monitoring, a construction of the automata from SEREs is described in [3]. The approach is very similar to the one presented here, using the same approach of handling symbolic labels on automata transitions. However, the automata and the translation are not formally presented. Unfeasible transitions are not removed, and no detail is given on how formulas are manipulated (BDD, DNF, or strings). The goal of the approach indeed is not the automata construction, but the generation of the circuit and the evaluation regards only the final hardware circuits. In particular, there is no comparison with standard libraries for automata manipulation.

5 Experimental Evaluation

The approach described in previous sections has been implemented in the AUTLIB library. The library is written in C, using adjacency lists to represent transitions outgoing from a state, and BDDs from the CUDD package (`http://vlsi.colorado.edu/~fabio`) for transition labels. The architecture is extensible to other forms of Boolean reasoning, such as propositional satisfiability (SAT), and to SMT. AUTLIB is used at the core of an extension of the NUSMV [7] model checker able to deal with the PSL language, and, as explained in [8], it is used to generate the automata necessary for PSL verification.

Set up. The proposed algorithm is evaluated in terms of *construction time* of an automaton corresponding to a SERE, and *number of states* of the resulting automaton.

The AUTLIB library was evaluated in two modes, with and without reduction. In the first mode, the activation of reduction is controlled by a simple heuristic, namely reduction is run only after **|**, **&&** and **&** operators. For the comparison, we use a test suite of 1200 SEREs, obtained by randomly modifying patterns extracted from industrial case studies. The SEREs are combinations of concatenations where the top level operators are randomly chosen in $\{|, \&\&, \&, [*], [+]\}$. The concatenations combine atomic Boolean expressions, or repetition of Boolean expressions using **[*]** or **[+]**. The number of concatenated SEREs is randomly chosen in the range $[2, 10]$. We generated 12 different families of benchmarks, choosing a possible configuration of parameters. The parameters are the number of top-level operators (which ranges in $\{1, 2\}$), the depth of Boolean expressions (which ranges in $\{2, 3\}$) and the number of atomic propositions (which ranges in $\{8, 10, 15\}$). For each family we generated 100 random SEREs.

AUTLIB is compared against the GRAZ library [15] and MONA [10]. Also for GRAZ we considered two operating modes: with and without NFA reduction. We compared with MONA through the STRANGER library [21], that provides concatenation and star as additional functions, and minimizes the DFA after such operations.

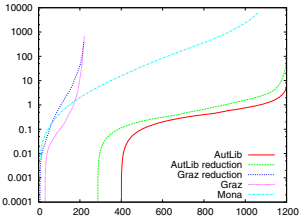
We ran the experiments on a Linux machine equipped with a 2.66GHz Intel(R) Core(TM)2 Quad Core, and 4GB of RAM with a time out of 120 seconds and memory limit of 3Gb. All results, together with the binaries and test cases necessary to reproduce them, are available at `http://es.fbk.eu/people/mover/tests/CIAA10/`.

Results. The results are presented in two different forms. Survival plots are used to provide a global view of the results: for each competitor, the "snake" shows the cumulative time required to solve a fixed number of instances. Pairwise comparison is obtained by means of scatter plots, where the x and y coordinates for each point represent the performance of the compared solvers on a given sample.

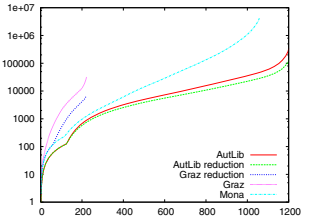
Figure 1(a) shows the cumulative plot for automata construction times for all the evaluated libraries. The GRAZ library, with and without reduction, shows poor performances, solving about 200 over 1200 examples. MONA can solve about 1050 examples while AUTLIB solves all the random generated SEREs. AUTLIB, with and without minimization, is much faster than MONA, and almost immediate on some instances.

AUTLIB and GRAZ can be compared from the scatter plots shown in Figure 1(c) and 1(d). All the examples where GRAZ can construct the automaton before timeout are trivial for AUTLIB. The bottleneck in the GRAZ approach is due to operations on labels performed on the DNF structure of formulas. Comparing the number of states, GRAZ generates bigger automata than AUTLIB. This is due to the management of labels in GRAZ, where for performance reasons the satisfiability check of a conjunction of labels is not complete. Transitions with inconsistent labels are thus created, possibly avoiding the pruning of unreachable states.

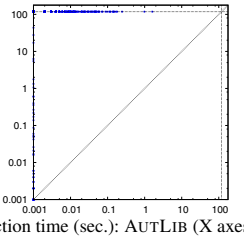
Figure 1(e) shows the scatter plot that compares construction times for MONA, on x axes, and AUTLIB, on y axes. AUTLIB outperforms MONA on every example. These results can be explained looking at Figure 1(f), that shows the number of states for the constructed automata. It is not surprising that DFAs generated by MONA are much bigger than NFASL of AUTLIB, since non-deterministic automata have a much succinct representation, which better adapts to common SERE expressions.



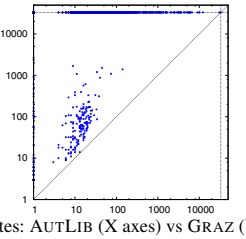
(a) Construction time (sec.)



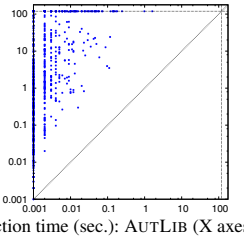
(b) Number of states



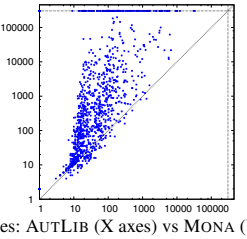
(c) Construction time (sec.): AUTLIB (X axes) vs GRAZ (Y axes)



(d) # of states: AUTLIB (X axes) vs GRAZ (Y axes)



(e) Construction time (sec.): AUTLIB (X axes) vs MONA (Y axes)



(f) # of states: AUTLIB (X axes) vs MONA (Y axes)

We also tested the effect of reductions for AUTLIB and GRAZ. For AUTLIB the benefits deriving from reductions, i.e., a reduced number of states, seem to be modest with respect to an increased construction time. As for GRAZ, although the reduction is a costly operation in terms of time, the benefits in the number of states are more evident. For lack of space, the scatter plots of the reductions are reported in the extended version of the paper.

6 Conclusions and Future Work

In this paper we have addressed the problem of providing automata-based techniques suitable for the manipulation of regular expressions arising in specification languages such as PSL. We propose an approach where non-deterministic finite automata are equipped with fully symbolic labels, represented by means of BDDs, over a given set of variables. We implemented an efficient library where all the standard functionalities are provided. The experiments demonstrate the need for the compactness of non-deterministic finite automata (compared to approaches based on deterministic finite automata), and the efficiency of a fully symbolic approach to label representation (with respect to an approach based on sets of partial assignments).

In the future, we plan to extend the experimentation with additional benchmarks, and to pinpoint possible bottlenecks of the current implementation. We will also investigate

the use of alternative symbolic technique (e.g. SAT and SMT solvers), and will develop fully symbolic minimization procedures.

Acknowledgments. We thank I. Pill for support with the GRAZ library, and O. Ibarra, T. Bultan, F. Yu and M. Alkhalaf for providing us with the STRANGER library.

References

1. IEEE Standard for Property Specification Language (PSL). IEEE Std 1850-2005 (2005)
2. Aziz Abdulla, P., Deneux, J., Kaati, L., Nilsson, M.: Minimization of non-deterministic automata with large alphabets. In: Farré, J., Litovsky, I., Schmitz, S. (eds.) CIAA 2005. LNCS, vol. 3845, pp. 31–42. Springer, Heidelberg (2006)
3. Boule, M., Zilic, Z.: Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation. In: ASP-DAC, pp. 324–329 (2007)
4. Champarnaud, J.-M.: Evaluation of Three Implicit Structures to Implement Nondeterministic Automata From Regular Expressions. Int. J. Found. Comput. Sci. 13(1), 99–113 (2002)
5. Champarnaud, J.M., Hansel, G.: Automate, a computing package for automata and finite semigroups. J. Symb. Comput. 12(2), 197–220 (1991)
6. Champarnaud, J.-M., Ponty, J.-L., Ziadi, D.: From Regular Expressions to Finite Automata. International Journal of Computer Mathematics 72(4), 415–431 (1999)
7. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A New Symbolic Model Checker. STTT 2(4), 410–425 (2000)
8. Cimatti, A., Roveri, M., Tonetta, S.: Symbolic Compilation of PSL. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1737–1750 (2008)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
10. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice (1995)
11. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
12. Ilie, L., Navarro, G., Yu, S.: On NFA Reductions. In: Theory is Forever, pp. 112–124 (2004)
13. Kell, V., Maier, A., Potthoff, A., Thomas, W., Wermuth, U.: AMORE: a system for computing automata, monoids and regular expressions. In: Cori, R., Monien, B. (eds.) STACS 1989. LNCS, vol. 349, pp. 537–538. Springer, Heidelberg (1989)
14. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems, Specification. Springer Verlag, New York (1992)
15. Pill, I.: Requirements Engineering and Efficient Verification of PSL properties. PhD thesis, Graz Univeristy of Technology (2008)
16. Raymond, D., Wood, D.: Grail: a C++ library for automata and expressions. J. Symb. Comput. 17(4), 341–350 (1994)
17. van Noord, G., Gerdemann, D.: Finite State Transducers with Predicates and Identities. Grammars 4(3), 263–286 (2001)
18. Veanes, M., Grigorenko, P., de Halleux, P., Tillmann, N.: Rex: Symbolic Regular Expression Explorer. In: ICST (2010)
19. Vijayaraghavan, S., Ramanathan, M.: A Practical Guide for SystemVerilog Assertions. Springer, Heidelberg (2005)
20. Watson, B.W.: A Taxonomy of Finite Automata Construction Algorithms. Technical report, Eindohoven University of Technology – Mathematics and Computing Science (1994)
21. Yu, F., Bultan, T., Cova, M., Ibarra, O.H.: Symbolic String Verification: An Automata-Based Approach. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 306–324. Springer, Heidelberg (2008)

State Complexity of Catenation Combined with Union and Intersection*

Bo Cui, Yuan Gao, Lila Kari, and Sheng Yu

Department of Computer Science,
The University of Western Ontario,
London, Ontario, Canada N6A 5B7

Abstract. In this paper, we study the state complexities of two particular combinations of operations: catenation combined with union and catenation combined with intersection. We show that the state complexity of the former combined operation is considerably less than the mathematical composition of the state complexities of catenation and union, while the state complexity of the latter one is equal to the mathematical composition of the state complexities of catenation and intersection.

1 Introduction

State complexity is a type of descriptional complexity for regular languages based on the deterministic finite automaton (DFA) model [18]. The state complexity of an operation on regular languages is the number of states that are necessary and sufficient in the worst case for the minimal, complete DFA that accepts the resulting language of the operation [6]. Many results on the state complexities of individual operations have been obtained, e.g. union, intersection, catenation, star, etc [1, 2, 7, 9, 10, 13, 14, 16, 18].

However, in practice, the operation to be performed is often a combination of several individual operations in a certain order, rather than only one individual operation. The study of state complexities of combined operations was initiated by A. Salomaa, K. Salomaa and S. Yu in 2007 [15] and followed by a number of papers [2–4, 11, 12]. It has been shown that the state complexity of a combined operation is not simply a mathematical composition of the state complexities of its component operations. It appears that the state complexity of a combined operation in general is more difficult to obtain than that of an individual operation, especially the tight lower bound of the operation. This is because the resulting languages of the worst case of one operation may not be among the worst case input languages of the subsequent operation.

The study of the state complexity of individual operations has already much relied on computer software to test and verify the results. One could say that,

* All correspondence should be directed to Yuan Gao at ygao72@csd.uwo.ca. This work is supported by Natural Science and Engineering Council of Canada Discovery Grant R2824A01, Canada Research Chair Award, and Natural Science and Engineering Council of Canada Discovery Grant 41630.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 95–104, 2011.

without the use of computer software, there would be no results on the state complexity of combined operations.

Although there is only a limited number of individual operations, the number of combined operations is unlimited. It is impossible to study the state complexity of all the combined operations. However, we consider that, besides the study of estimation and approximation of state complexity of general combined operations [4, 5], it is important that the exact state complexity of some commonly used and basic combined operations should be studied.

In this paper, we study the state complexities of catenation combined with union, i.e., $(L(A)(L(B)\cup L(C)))$, and catenation combined with intersection, i.e., $(L(A)(L(B)\cap L(C)))$, for DFAs A, B and C of sizes $m, n, p \geq 1$, respectively. Both of them are basic combined operations and are commonly used in practice. For $L(A)(L(B)\cup L(C))$, we show that its state complexity is $(m-1)(2^{n+p}-2^n-2^p+2)+2^{n+p-2}$, for $m, n, p \geq 1$ (except the situations when $m \geq 2$ and $n = p = 1$), which is much smaller than $m2^{np}-2^{np-1}$, the mathematical composition of the state complexities of union and catenation [13, 16]. On the other hand, for $L(A)(L(B)\cap L(C))$, we show that the mathematical composition of the individual state complexities of this combined operation is $m2^{np}-2^{np-1}$, i.e., exactly equal to the state complexity of the operation (also except the cases when $m \geq 2$ and $n = p = 1$). Note that the individual state complexity of union and that of intersection are exactly the same. However, when they combined with catenation, the resulting state complexities are so different.

In the next section, we introduce the basic definitions and notations used in the paper. Then we prove our results on catenation combined with union and catenation combined with intersection in Sections 3 and 4, respectively. We conclude the paper in Section 5.

2 Preliminaries

A *non-deterministic finite automaton* (NFA) is a quintuple $A = (Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, $s \in Q$ is the start state, and $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^Q$ is the transition function. If $|\delta(q, a)| \leq 1$ for any $q \in Q$ and $a \in \Sigma$, then this automaton is called a *deterministic finite automaton* (DFA). A DFA is said to be complete if $|\delta(q, a)| = 1$ for all $q \in Q$ and $a \in \Sigma$. All the DFAs we mention in this paper are assumed to be complete. We extend δ to $Q \times \Sigma^* \to Q$ in the usual way. Then a word $w \in \Sigma^*$ is accepted by the automaton if $\delta(s, w) \cap F \neq \emptyset$. Two states in a finite automaton A are said to be *equivalent* if and only if for every word $w \in \Sigma^*$, if A is started in either state with w as input, it either accepts in both cases or rejects in both cases. It is well-known that a language which is accepted by an NFA can be accepted by a DFA, and such a language is said to be *regular*. The language accepted by a DFA A is denoted by $L(A)$. The reader may refer to [8, 17] for more details about regular languages and finite automata.

The *state complexity* of a regular language L, denoted by $sc(L)$, is the number of states of the minimal complete DFA that accepts L. The state complexity of

a class S of regular languages, denoted by $sc(S)$, is the supremum among all $sc(L)$, $L \in S$. The state complexity of an operation on regular languages is the state complexity of the resulting languages from the operation as a function of the state complexity of the operand languages. For example, we say that the state complexity of the intersection of an m-state DFA language and an n-state DFA language is exactly mn. This implies that the largest number of states of all the minimal complete DFAs that accept the intersection of an m-state DFA language and an n-state DFA language is mn, and such languages exist. Thus, in a certain sense, the state complexity of an operation is a worst-case complexity.

3 Catenation Combined with Union

In this section, we consider the state complexity of $L(A)(L(B) \cup L(C))$ for three DFAs A, B, C of sizes $m, n, p \geq 1$, respectively. We first obtain the following upper bound $(m-k)(2^{n+p} - 2^n - 2^p + 2) + k2^{n+p-2}$ (Theorem 1), and then show that this bound is tight for $m, n, p \geq 1$, except the situations when $m \geq 2$ and $n = p = 1$ (Theorems 2 and 3).

Theorem 1. *For integers $m, n, p \geq 1$, let A, B and C be three DFAs with m, n and p states, respectively, where A has k final states. Then, there exists a DFA of at most $(m-k)(2^{n+p} - 2^n - 2^p + 2) + k2^{n+p-2}$ states that accepts $L(A)(L(B) \cup L(C))$.*

Proof. Let $A = (Q_1, \Sigma, \delta_1, s_1, F_1)$ where $|F_1| = k$, $B = (Q_2, \Sigma, \delta_2, s_2, F_2)$, and $C = (A_3, \Sigma, \delta_3, s_3, F_3)$. We construct $D = (Q, \Sigma, \delta, s, F)$ such that

$$
\begin{aligned}
Q = {} & \{\langle q_1, q_2, q_3\rangle \mid q_1 \in Q_1 - F_1, q_2 \in 2^{Q_2} - \{\emptyset\}, q_3 \in 2^{Q_3} - \{\emptyset\}\} \\
& \cup \{\langle q_1, \emptyset, \emptyset\rangle \mid q_1 \in Q_1 - F_1\} \\
& \cup \{\langle q_1, \{s_2\} \cup q_2, \{s_3\} \cup q_3\rangle \mid q_1 \in F_1, q_2 \in 2^{Q_2 - \{s_2\}}, q_3 \in 2^{Q_3 - \{s_3\}}\}, \\
s = {} & \langle s_1, \emptyset, \emptyset\rangle \text{ if } s_1 \notin F_1, s = \langle s_1, \{s_2\}, \{s_3\}\rangle \text{ otherwise}, \\
F = {} & \{\langle q_1, q_2, q_3\rangle \in Q \mid q_2 \cap F_2 \neq \emptyset \text{ or } q_3 \cap F_3 \neq \emptyset\}, \\
\delta(\langle q_1, q_2, q_3\rangle, a) = {} & \langle q_1', q_2', q_3'\rangle, \text{ for } a \in \Sigma, \text{ where } q_1' = \delta_1(q_1, a) \text{ and}, \\
& \text{for } i \in \{2, 3\}, q_i' = S_i \cup \{s_i\} \text{ if } q_1' \in F_1, q_i' = S_i \text{ otherwise}, \\
& \text{where } S_i = \cup_{r \in q_i} \{\delta_i(r, a)\}.
\end{aligned}
$$

Intuitively, Q is a set of triples such that the first component of each triple is a state in Q_1 and the second and the third components are subsets of Q_2 and Q_3, respectively.

We notice that if the first component of a state is a non-final state of Q_1, the other two component are either both the empty set or both nonempty sets. This is because the two components always change from the empty set to a non-empty set at the same time. This is the reason to have the first and second terms of Q.

Also, we notice that if the first component of a state of D is a final state of A, then the second component and the third component of the state must contain the initial state of B and C, respectively. This is described by the third term of Q.

Clearly, the size of Q is $(m-k)(2^{n+p}-2^n-2^p+2)+k2^{n+p-2}$. Moreover, one can easily verify that $L(D)=L(A)(L(B)\cup L(C))$. □

In the following, we consider the conditions under which this bound is tight. We know that a complete DFA of size 1 only accepts either $\emptyset$ or Σ^*. Thus, when $n=p=1$, $L(A)(L(B)\cup L(C))=L(A)\Sigma^*$ if either $L(B)=\Sigma^*$ or $L(C)=\Sigma^*$, and $L(A)(L(B)\cup L(C))=\emptyset$ otherwise. Therefore, in such cases, the state complexity of $L(A)(L(B)\cup L(C))$ is m as shown in [16].

Now, we consider the case when $n=1$ and $p\geq 2$. Since $L(B)\cup L(C)=L(C)$ when $L(B)=\emptyset$, it is clear that the state complexity of $L(A)(L(B)\cup L(C))$ is equal to that of $L(A)L(C)$, $m2^p-k2^{p-1}$ given in [16], which coincides with the upper bound obtained in Theorem 1. The situation is analogous to the case when $n\geq 2$ and $p=1$.

Next, we consider the case when $m=1$ and $n,p\geq 2$.

Theorem 2. *Let A be a DFA of size 1. Then, for any integers $n,p\geq 2$, there exist DFAs B and C with n and p states, respectively, such that any DFA accepting $L(A)(L(B)\cup L(C))$ needs at least 2^{n+p-2} states.*

Proof. We use a four-letter alphabet $\Sigma=\{a,b,c,d\}$, and let A be the DFA accepting Σ^*.

Let $B=(Q_2,\Sigma,\delta_2,0,\{n-1\})$, shown in Figure 1, where $Q_2=\{0,1,\ldots,n-1\}$, and the transitions are given as

- $\delta_2(i,a)=i+1 \bmod n$, for $i\in\{0,\ldots,n-1\}$,
- $\delta_2(i,x)=i$ for $i\in Q_2$, where $x\in\{b,d\}$,
- $\delta_2(0,c)=0$, $\delta_2(i,c)=i+1 \bmod n$, for $i\in\{1,\ldots,n-1\}$.

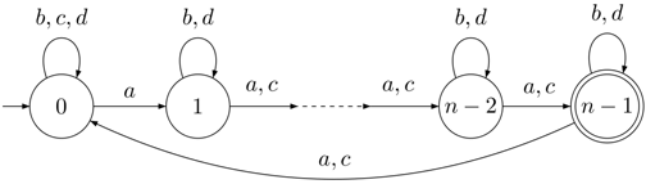


Fig. 1. DFA B used for showing that the upper bound in Theorem 1 is reachable when $m=1$ and $n,p\geq 2$

Let $C=(Q_3,\Sigma,\delta_3,0,\{p-1\})$, whose transition diagram is similar to the one shown in Figure 1, where $Q_3=\{0,1,\ldots,p-1\}$, and the transitions are given as

- $\delta_3(i,x)=i$ for $i\in Q_3$, where $x\in\{a,c\}$,
- $\delta_3(i,b)=i+1 \bmod p$, for $i\in\{0,\ldots,p-1\}$,
- $\delta_3(0,d)=0$, $\delta_3(i,d)=i+1 \bmod p$, for $i\in\{1,\ldots,p-1\}$.

Let $D = (Q, \{a, b, c, d\}, \delta, \langle 0, \{0\}, \{0\}\rangle, F)$ be the DFA for accepting the language $L(A)(L(B) \cup L(C))$ constructed from those DFAs exactly as described in the proof of the previous theorem, where

$$Q = \{\langle 0, \{0\} \cup q_2, \{0\} \cup q_3\rangle \mid q_2 \in 2^{Q_2 - \{0\}}, q_3 \in 2^{Q_3 - \{0\}}\},$$
$$F = \{\langle q_1, q_2, q_3\rangle \in Q \mid n-1 \in q_2 \text{ or } p-1 \in q_3\}.$$

We omit the definition of the transitions.

Then we prove that the size of Q 2^{n+p-2} is minimal by showing that (I) any state in Q can be reached from the initial state, and (II) no two different states in Q are equivalent.

For (I), we first show that all the states $\langle 0, q_2, q_3\rangle$ such that $q_3 = \{0\}$ are reachable by induction on the size of q_2.

The basis clearly holds, since the initial state is the only state whose second component is of size 1.

In the induction steps, we assume that all states $\langle 0, q_2, \{0\}\rangle$ such that $|q_2| < k$ are reachable. Then, we consider the states $\langle 0, q_2, \{0\}\rangle$ where $|q_2| = k$. Let $q_2 = \{0, j_2, \ldots, j_k\}$ such that $0 < j_2 < j_3 < \ldots < j_k \leq n-1$. Note that the states such that $j_2 = 1$ can be reached as follows

$$\langle 0, \{0, 1, j_3, \ldots, j_k\}, \{0\}\rangle = \delta(\langle 0, \{0, j_3 - 1, \ldots, j_k - 1\}, \{0\}\rangle, a),$$

where $\{0, j_3 - 1, \ldots, j_k - 1\}$ is of size $k-1$. Then, the states such that $j_2 > 1$ can be reached from these states as follows

$$\langle 0, \{0, j_2, \ldots, j_k\}, \{0\}\rangle = \delta(\langle 0, \{0, 1, j_3 - t, \ldots, j_k - t\}, \{0\}\rangle, c^t), \text{ where } t = j_2 - 1.$$

After this induction, all the states such that the third component is $\{0\}$ have been reached. Then, it is clear that, from each of these states $\langle 0, q_2, \{0\}\rangle$, all the states in Q such that the second component is q_2 and the size of their third component is larger than 1 can be reached by using the same induction steps but using the transitions on letters b and d.

Next, we show that any two distinct states $\langle 0, q_2, q_3\rangle$ and $\langle 0, q_2', q_3'\rangle$ in Q are not equivalent. We only consider the situations where $q_2 \neq q_2'$, since the other case can be shown analogously. Without loss of generality, there exists a state r such that $r \in q_2$ and $r \notin q_2'$. It is clear that $r \neq 0$. Let $w = d^{p-1}c^{n-1-r}$. Then $\delta(\langle 0, q_2, q_3\rangle, w) \in F$ but $\delta(\langle 0, q_2', q_3'\rangle, w) \notin F$. □

Then, we consider the more general case when $m, n, p \geq 2$.

Example 1. We use a five-letter alphabet $\Sigma = \{a, b, c, d, e\}$ in the following three DFAs, which are modified from the two DFAs in the proof of Theorem 1 in [16].

Let $A = (Q_1, \Sigma, \delta_1, 0, \{m-1\})$, where $Q_1 = \{0, \ldots, m-1\}$ and, for each state $i \in Q_1$, $\delta_1(i, a) = j$, $j = (i+1) \bmod m$, $\delta_1(i, x) = 0$, if $x \in \{b, d\}$, and $\delta_1(i, x) = i$, if $x \in \{c, e\}$.

Let $B = (Q_2, \Sigma, \delta_2, 0, \{n-1\})$, where $Q_2 = \{0, \ldots, n-1\}$ and, for each state $i \in Q_2$, $\delta_2(i, b) = j$, $j = (i+1) \bmod m$, $\delta_2(i, c) = 1$, and $\delta_2(i, x) = i$, if $x \in \{a, d, e\}$.

Let $C = (Q_3, \Sigma, \delta_3, 0, \{p-1\})$, where $Q_3 = \{0, \ldots, p-1\}$ and, for each state $i \in Q_3$, $\delta_3(i,d) = j$, $j = (i+1) \bmod m$, $\delta_3(i,e) = 1$, and $\delta_3(i,x) = i$, if $x \in \{a,b,c\}$.

Following the construction in the proof of Theorem 1, a DFA D can be constructed from the DFAs in Example 1 for showing that the upper bound is reachable for $m,n,p \geq 2$. We note that, similar to the proof of Theorem 2, DFAs B and C in this example change their states on disjoint letter sets, $\{b,c\}$ and $\{d,e\}$. Thus, by using a proof that is similar to the proof of Theorem 1 in [16], that shows the upper bound of the state complexity of catenation can be reached, we can easily verify that there are at least $(m-1)(2^{n+p}-2^n-2^p+2)+2^{n+p-2}$ distinct equivalence classes of the right-invariant relation induced by $L(A)(L(B) \cup L(C))$ [8]. Therefore, the upper bound can be reached and the following theorem holds.

Theorem 3. *Given three integers $m,n,p \geq 2$, there exist a DFA A of m states, a DFA B of n states, and a DFA C of p states such that any DFA accepting $L(A)(L(B) \cup L(C))$ needs at least $(m-1)(2^{n+p}-2^n-2^p+2)+2^{n+p-2}$ states.*

A natural question is that, if we reduce the size of the alphabet used in DFAs A, B, C, using a three-letter alphabet, can we reach the upper bound as well? We give a positive answer in the next theorem under the condition $m,n,p \geq 3$.

Theorem 4. *For any integers $m,n,p \geq 3$, there exist DFAs A, B and C of m, n, and p states, respectively, defined over a three-letter alphabet, such that any DFA accepting $L(A)(L(B) \cup L(C))$ needs at least $(m-1)(2^{n+p}-2^n-2^p+2)+2^{n+p-2}$ states.*

4 Catenation Combined with Intersection

In this section, we investigate the state complexity of $L_1(L_2 \cap L_3)$, and show that its upper bound (Theorem 5) coincides with its lower bound (Theorems 6 and 7). The following theorem shows an upper bound of the state complexity of this combined operation.

Theorem 5. *Let L_1, L_2 and L_3 be three regular languages accepted by an m-state, an n-state and a p-state DFA, respectively, for m, n, $p \geq 1$. Then there exists a DFA of at most $m2^{np}-2^{np-1}$ states that accepts $L_1(L_2 \cap L_3)$.*

We omit the proof of Theorem 5 because $m2^{np}-2^{np-1}$ is the mathematical composition of the state complexities of the individual component operations, which is obviously an upper bound of the state complexity of $L_1(L_2 \cap L_3)$. In the following, we investigate the lower bounds of the state complexity of this combined operation under different conditions.

When $n = p = 1$, $L(A)(L(B) \cap L(C)) = L(A)\Sigma^*$ if both $L(B)$ and $L(C)$ are Σ^*. The resulting language is $\emptyset$ otherwise. Thus, the state complexity of $L(A)(L(B) \cap L(C))$ in this case is the same as that of $L(A)\Sigma^*$: m [16].

When $n = 1$, $p \geq 2$, $L(A)(L(B) \cap L(C)) = \emptyset$, if $L(B) = \emptyset$, and $L(A)L(C)$ if $L(B) = \Sigma^*$. In this case, the state complexity of the combined operation is

$m2^p - 2^{p-1}$ which is the same as that of $L(A)L(C)$ [16]. Similarly, when $n \geq 2$, $p = 1$, the state complexity of $L(A)(L(B) \cap L(C))$ is $m2^n - 2^{n-1}$. Next, we show the upper bound $m2^{np} - 2^{np-1}$ is reachable when $m, n, p \geq 2$.

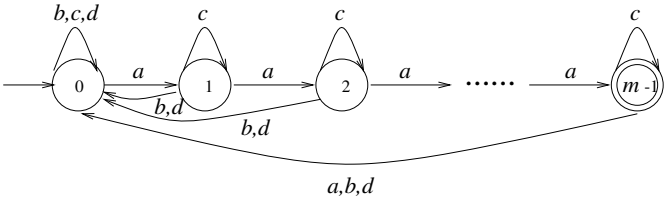


Fig. 2. DFA A used for showing that the upper bound in Theorem 5 is reachable when $m \geq 2$ and $n, p \geq 1$

Theorem 6. *Given three integers $m, n, p \geq 2$, there exists a DFA A of m states, a DFA B of n states and a DFA C of p states such that any DFA accepting $L(A)(L(B) \cap L(C))$ needs at least $m2^{np} - 2^{np-1}$ states.*

Proof. Let $A = (Q_A, \Sigma, \delta_A, 0, F_A)$ be a DFA, shown in Figure 2, where $Q_A = \{0, 1, \ldots, m-1\}$, $F_A = \{m-1\}$, $\Sigma = \{a, b, c, d\}$ and the transitions are given as:

- $\delta_A(i, a) = i + 1 \bmod m$, $i = 0, \ldots, m-1$,
- $\delta_A(i, x) = 0$, $i = 0, \ldots, m-1$, where $x \in \{b, d\}$,
- $\delta_A(i, c) = i$, $i = 0, \ldots, m-1$.

Let $B = (Q_B, \Sigma, \delta_B, 0, F_B)$ be a DFA, shown in Figure 3, where $Q_B = \{0, 1, \ldots, n-1\}$, $F_B = \{n-1\}$ and the transitions are given as:

- $\delta_B(i, x) = i$, $i = 0, \ldots, n-1$, where $x \in \{a, d\}$,
- $\delta_B(i, b) = i + 1 \bmod n$, $i = 0, \ldots, n-1$,
- $\delta_B(i, c) = 1$, $i = 0, \ldots, n-1$.

Let $C = (Q_C, \Sigma, \delta_C, 0, F_C)$ be a DFA, whose transition diagram is similar to the one shown in Figure 3, where $Q_C = \{0, 1, \ldots, p-1\}$, $F_C = \{p-1\}$ and the transitions are given as:

- $\delta_C(i, x) = i$, $i = 0, \ldots, p-1$, where $x \in \{a, b\}$,
- $\delta_C(i, c) = 1$, $i = 0, \ldots, p-1$,
- $\delta_C(i, d) = i + 1 \bmod p$, $i = 0, \ldots, p-1$.

We construct a DFA $D = (Q_D, \Sigma, \delta_D, s_D, F_D\}$, where

$$
\begin{aligned}
Q_D &= \{\langle u, v\rangle \mid u \in Q_B,\ v \in Q_C\},\\
s_D &= \langle 0, 0\rangle,\\
F_D &= \{\langle n-1, p-1\rangle\},
\end{aligned}
$$

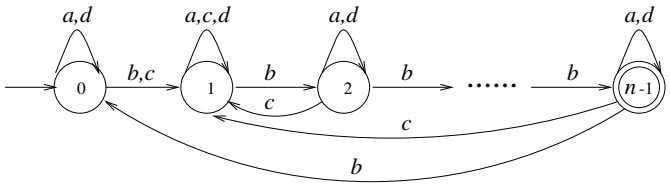


Fig. 3. DFA B used for showing that the upper bound in Theorem 5 is reachable when $m \geq 2$ and $n, p \geq 1$

and for each state $\langle u, v\rangle \in Q_D$ and each letter $e \in \Sigma$,

$$\delta_D(\langle u, v\rangle, e) = \langle u', v'\rangle \text{ if } \delta_B(u, e) = u',\ \delta_C(v, e) = v'.$$

Clearly, there are $n \cdot p$ states in D and $L(D) = L(B) \cap L(C)$. Now we construct another DFA $E = (Q_E, \Sigma, \delta_E, s_E, F_E\}$, where

$$\begin{aligned} Q_E &= \{\langle q, R\rangle \mid q \in Q_A - F_A,\ R \subseteq Q_D\} \cup \{\langle m-1, S\rangle \mid s_D \in S,\ S \subseteq Q_D\},\\ s_E &= \langle 0, \emptyset\rangle,\\ F_E &= \{\langle q, R\rangle \mid R \cap F_D \neq \emptyset,\ \langle q, R\rangle \in Q_E\}, \end{aligned}$$

and for each state $\langle q, R\rangle \in Q_E$ and each letter $e \in \Sigma$,

$$\delta_E(\langle q, R\rangle, e) = \begin{cases} \langle q', R'\rangle \text{ if } \delta_A(q, e) = q' \neq m-1,\ \delta_D(R, e) = R',\\ \langle q', R'\rangle \text{ if } \delta_A(q, e) = q' = m-1,\ R' = \delta_D(R, e) \cup \{s_D\}. \end{cases}$$

It is easy to see that $L(E) = L(A)(L(B) \cap L(C))$. There are $(m-1) \cdot 2^{np}$ states in the first term of the union for Q_E. In the second term, there are $1 \cdot 2^{np-1}$ states. Thus,

$$|Q_E| = (m-1) \cdot 2^{np} + 1 \cdot 2^{np-1} = m2^{np} - 2^{np-1}.$$

In order to show that E is minimal, we need to show that (I) every state in E is reachable from the start state and (II) each state defines a distinct equivalence class.

We prove (I) by induction on the size of the second component of states in Q_E. First, any state $\langle q, \emptyset\rangle$, $0 \leq q \leq m-2$, is reachable from s_E by reading a word a^q. The we consider all states $\langle q, R\rangle$ such that $|R| = 1$. In this case, let $R = \{\langle x, y\rangle\}$. We have

$$\langle q, \{\langle x, y\rangle\}\rangle = \delta_E(\langle 0, \emptyset\rangle, a^m b^x d^y a^q).$$

Notice that the only state $\langle q, R\rangle$ in Q_E such that $q = m-1$ and $|R| = 1$ is $\langle m-1, \{\langle 0, 0\rangle\}\rangle$ since the fact that $q = m-1$ guarantees $\langle 0, 0\rangle \in R$.

Assume that all states $\langle q, R\rangle$ such that $|R| < k$ are reachable. Consider $\langle q, R\rangle$ where $|R| = k$. Let $R = \{\langle x_i, y_i\rangle \mid 1 \leq i \leq k\}$ such that $0 \leq x_1 \leq x_2 \leq \ldots \leq x_k \leq n-1$ if $q \neq m-1$ and $0 = x_1 \leq x_2 \leq \ldots \leq x_k \leq n-1$, $y_1 = 0$, otherwise. We have $\langle q, R\rangle = \delta_E(\langle 0, R'\rangle, a^m b^{x_1} d^{y_1} a^q)$, where

$$R' = \{\langle x_j - x_1, (y_j - y_1 + n) \bmod n\rangle \mid 2 \leq j \leq k\}.$$

State $\langle 0, R' \rangle$ is reachable from the start state, since $|R| = k - 1$. Thus, $\langle q, R \rangle$ is also reachable.

To prove (II), let $\langle q_1, R_1 \rangle$ and $\langle q_2, R_2 \rangle$ be two different states in E. We consider the following two cases.

1. $q_1 \neq q_2$. Without loss of generality, we may assume that $q_1 > q_2$. There always exists a string $t = ca^{m-1-q_1}b^{n-1}d^{p-1}$ such that
$$\delta_E(\langle q_1, R_1 \rangle, t) \in F_E,$$
$$\delta_E(\langle q_2, R_2 \rangle, t) \notin F_E.$$
2. $q_1 = q_2$, $R_1 \neq R_2$. Without loss of generality, we may assume that $|R_1| \geq |R_2|$. Let $\langle x, y \rangle \in R_1 - R_2$. Then
$$\delta_E(\langle q_1, R_1 \rangle, b^{n-1-x}d^{p-1-y}) \in F_E,$$
$$\delta_E(\langle q_2, R_2 \rangle, b^{n-1-x}d^{p-1-y}) \notin F_E.$$

Thus, the minimal DFA accepting $L(A)(L(B) \cap L(C))$ needs at least $m2^{np} - 2^{np-1}$ states for $m, n, p \geq 2$. □

Now we consider the case when $m = 1$, i.e., $L(A) = \Sigma^*$.

Theorem 7. *Given two integers $n, p \geq 2$, there exists a DFA A of 1 state, a DFA B of n states and a DFA C of p states such that any DFA accepting $L(A)(L(B) \cap L(C))$ needs at least 2^{np-1} states.*

This lower bound coincides with the upper bound given in Theorem 5. Thus, the bounds are tight for the case when $m = 1$, $n, p \geq 2$.

5 Conclusion

In this paper, we have studied the state complexities of two basic combined operations: catenation combined with union and catenation combined with intersection. We have proved that the state complexity of $L(A)(L(B) \cup L(C))$ is $(m-1)(2^{n+p} - 2^n - 2^p + 2) + 2^{n+p-2}$ for $m, n, p \geq 1$ (except the situations when $m \geq 2$ and $n = p = 1$), which is significantly less than the mathematical composition of state complexities of its component operations, $m2^{np} - 2^{np-1}$. We have also proved that the state complexity of $L(A)(L(B) \cap L(C))$ is $m2^{np} - 2^{np-1}$ for $m, n, p \geq 1$ (except the cases when $m \geq 2$ and $n = p = 1$), which is exactly the mathematical composition of state complexities of its component operations. An interesting question is: why are the state complexity results on these two very similar combined operations so different?

Acknowledgement

We would like to thank the anonymous referees of CIAA 2010 for their careful reading and valuable suggestions.

References

1. Campeanu, C., Culik, K., Salomaa, K., Yu, S.: State complexity of basic operations on finite language. In: Boldt, O., Jürgensen, H. (eds.) WIA 1999. LNCS, vol. 2214, pp. 60–70. Springer, Heidelberg (2001)
2. Domaratzki, M., Okhotin, A.: State complexity of power. Theoretical Computer Science 410(24-25), 2377–2392 (2009)
3. Ésik, Z., Gao, Y., Liu, G., Yu, S.: Estimation of state complexity of combined operations. Theoretical Computer Science 410(35), 3272–3280 (2009)
4. Gao, Y., Salomaa, K., Yu, S.: The state complexity of two combined operations: star of catenation and star of Reversal. Fundam. Inform. 83(1-2), 75–89 (2008)
5. Gao, Y., Yu, S.: State complexity approximation. In: Proceedings of Descriptional Complexity of Formal Systems, pp. 163–174 (2009)
6. Han, Y., Salomaa, K.: State complexity of basic operations on suffix-free regular languages. Theoretical Computer Science 410(27-29), 2537–2548 (2009)
7. Holzer, M., Kutrib, M.: State complexity of basic operations on nondeterministic finite automata. In: Champarnaud, J.-M., Maurel, D. (eds.) CIAA 2002. LNCS, vol. 2608, pp. 148–157. Springer, Heidelberg (2003)
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, Reading (1979)
9. Jirásek, J., Jirásková, G., Szabari, A.: State complexity of concatenation and complementation of regular languages. International Journal of Foundations of Computer Science 16, 511–529 (2005)
10. Jirásková, G.: State complexity of some operations on binary regular languages. Theoretical Computer Science 330, 287–298 (2005)
11. Jirásková, G., Okhotin, A.: On the state complexity of star of union and star of intersection. Turku Center for Computer Science TUCS Technical Report No. 825 (2007)
12. Liu, G., Martin-Vide, C., Salomaa, A., Yu, S.: State complexity of basic language operations combined with reversal. Information and Computation 206, 1178–1186 (2008)
13. Maslov, A.N.: Estimates of the number of states of finite automata. Soviet Mathematics Doklady 11, 1373–1375 (1970)
14. Salomaa, A., Wood, D., Yu, S.: On the state complexity of reversals of regular languages. Theoretical Computer Science 320, 293–313 (2004)
15. Salomaa, A., Salomaa, K., Yu, S.: State complexity of combined operations. Theoretical Computer Science 383, 140–152 (2007)
16. Yu, S., Zhuang, Q., Salomaa, K.: The state complexity of some basic operations on regular languages. Theoretical Computer Science 125, 315–328 (1994)
17. Yu, S.: Regular languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 1, pp. 41–110. Springer, Heidelberg (1997)
18. Yu, S.: State complexity of regular languages. Journal of Automata, Languages and Combinatorics 6(2), 221–234 (2001)

Complexity Results and the Growths of Hairpin Completions of Regular Languages (Extended Abstract)

Volker Diekert and Steffen Kopecki

Universität Stuttgart, FMI
Universitätsstr. 38, 70569 Stuttgart, Germany
{diekert,kopecki}@fmi.uni-stuttgart.de

Abstract. The hairpin completion is a natural operation on formal languages which has been inspired by molecular phenomena in biology and by DNA-computing. In 2009 we presented in [6] a (polynomial time) decision algorithm to decide regularity of the hairpin completion. In this paper we provide four new results: 1.) We show that the decision problem is **NL**-complete. 2.) There is a polynomial time decision algorithm which runs in time $\mathcal{O}(n^8)$, this improves [6], which provided $\mathcal{O}(n^{20})$. 3.) For the one-sided case (which is closer to DNA computing) the time is $\mathcal{O}(n^2)$, only. 4.) The hairpin completion is unambiguous linear context-free. This result allows to compute the growth (generating function) of the hairpin completion and to compare it with the growth of the underlying regular language.

1 Introduction

The hairpin completion is a natural operation of formal languages which has been inspired by molecular phenomena in biology and by DNA-computing. An intramolecular base pairing, known as a *hairpin*, is a pattern that can occur in single-stranded DNA and, more commonly, in RNA. Hairpin or hairpin-free structures have numerous applications to DNA computing and molecular genetics, see [5,8,9,13,14] and the references within. For example, an instance of 3-Sat has been solved with a DNA-algorithm and one of the main concepts was to eliminate all molecules with a hairpin structure, see [18].

In this paper we study the hairpin completion from a purely formal language viewpoint. The hairpin completion of a formal language was first defined in [4]; here we use a slightly more general definition which was introduced in [6]. The formal operation of the hairpin completion on words is best explained in Fig. 1. In that picture as in the rest of the paper we mean by putting a *bar* on a word (like $\overline{\alpha}$) to read it from right-to-left in addition to replacing a with the *Watson-Crick complement* $\overline{a}$ for letters. The hairpin completion of a regular language is linear context-free [4]. For some time it was not known whether regularity of the hairpin completion is decidable. It was only in 2009 when we presented in [6] a decision algorithm. The runtime of that algorithm is in $\mathcal{O}(n^{20})$, hence polynomial.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 105–114, 2011.

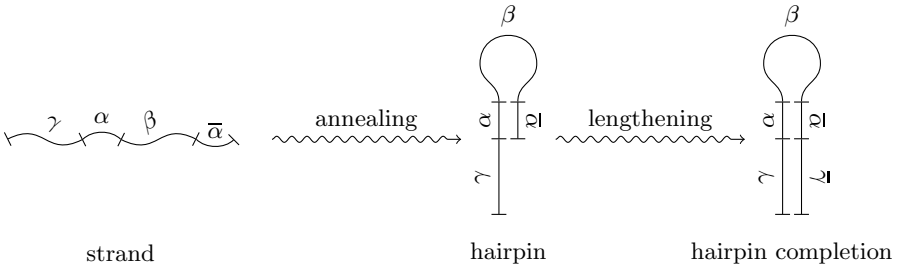


Fig. 1. Hairpin completion of a DNA-strand (or a word)

Here we present a modified approach to solve the decision problem. The new approach leads to improved complexity results and a new structure theorem. We show that the decision problem is **NL**-complete (Thm. 1). We show that there is a polynomial time decision algorithm which runs in time $\mathcal{O}(n^8)$ (Thm. 2, i.). So, the improvement is from $\mathcal{O}(n^{20})$ down to $\mathcal{O}(n^8)$. Moreover, in the biological model the one-sided hairpin completion is of particular interest, and in that special case we need quadratic time, only (Thm. 2, iii.). We also argue why the time bounds might be optimal in the worst case.

A byproduct of the method yields that the hairpin completion of a regular language is unambiguous linear context-free (Thm. 4). The result about unambiguity allows to compute the growth (generating function) of the hairpin completion and to compare it with the growth of the underlying regular language (Thm. 3 and Cor. 1).

This takes us back to a challenging open problem in formal languages. Regularity of linear context-free languages is undecidable in general [1,11]. But the situation for unambiguous context-free languages is open for more than 40 years. Hence, we have now a positive result within the classical context of deciding regularity within a class of unambiguous (linear) context-free languages.

Due to page limitation, some of the proofs have been removed. The missing proofs can be found in the technical report [7].

2 Preliminaries and Notation

We assume the reader to be familiar with the fundamental concepts of formal language theory, automata theory, and complexity theory, see [12,17]. By **NL** we mean the complexity class NLOGSPACE, which contains the problems which can be decided with a non-deterministic algorithm using $\mathcal{O}(\log n)$ space. We heavily rely on the well-known result that **NL** is closed under complementation. We also use the fact that if L can be reduced to L' via some single-valued non-deterministic transduction in $\mathcal{O}(\log n)$ space and $L' \in \mathbf{NL}$, then we have $L \in \mathbf{NL}$, too. This reduction is performed by a non-deterministic log-space Turing machine. In case the machine stops on input w, the output is always the same, independently of non-deterministic moves during the computation. So, we can

call the output $r(w)$. The reduction property tells us $w \in L$ if and only if both, the machine sometimes stops on input w and $r(w) \in L'$.

By Σ we denote a finite alphabet with at least two letters which is equipped with an *involution* $\overline{\ } : \Sigma \to \Sigma$. An involution for a set is a bijection such that $\overline{\overline{a}} = a$. We extend the involution to words $a_1 \cdots a_n$ by $\overline{a_1 \cdots a_n} = \overline{a_n} \cdots \overline{a_1}$. (Just like taking inverses in groups.) For languages $\overline{L}$ denotes the set $\{\overline{w} \mid w \in L\}$. The set of words over Σ is denoted Σ^*; and the *empty word* is denoted by 1. Given a word w, we denote by $|w|$ its length and $w(m) \in \Sigma$ its m-th letter. By $\Sigma^{\leq m}$ we mean the set of all words with length at most m. If $w = xyz$ for some $x, y, z \in \Sigma^*$, then x and z are called *prefix* and *suffix*, respectively. The prefix relation between words x and w is denoted by $x \leq w$.

Throughout the paper L_1, L_2 mean two regular languages in Σ^* and by k we mean a (small) constant, say $k = 10$. We define the *hairpin completion* $\mathcal{H}_k(L_1, L_2)$ by

$$\mathcal{H}_k(L_1, L_2) = \{\gamma\alpha\beta\overline{\alpha}\overline{\gamma} \mid (\gamma\alpha\beta\overline{\alpha} \in L_1 \vee \alpha\beta\overline{\alpha}\overline{\gamma} \in L_2) \wedge |\alpha| = k\}\,.$$

Three cases are of main interest: 1.) $L_1 = L_2$, 2.) $L_1 = \overline{L_2}$, and 3.) $L_1 = \emptyset$ or $L_2 = \emptyset$. Compared to the definition of the hairpin completion in [4,16] case 1.) corresponds to the the two-sided hairpin completion and case 3.) to the one-sided hairpin completion. Since we have better time complexities for 2.) and 3.) than for 1.) or in the general case we make the time bounds rather precise.

Regular languages can be specified by non-deterministic finite automata (NFA) $\mathcal{A} = (\mathcal{Q}, \Sigma, E, \mathcal{I}, \mathcal{F})$, where $\mathcal{Q}$ is the finite set of *states*, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of *initial states*, and $\mathcal{F} \subseteq \mathcal{Q}$ is the set of *final states*. The set E contains labeled *edges* (or *arcs*), it is a subset of $\mathcal{Q} \times \Sigma \times \mathcal{Q}$. For a word $u \in \Sigma^*$ we write $p \xrightarrow{u} q$, if there is a path from state p to q which is labeled by the word u. Thus, the accepted language becomes

$$L(\mathcal{A}) = \left\{ u \in \Sigma^* \;\middle|\; \exists p \in \mathcal{I},\, \exists q \in \mathcal{F} :\, p \xrightarrow{u} q \right\}.$$

Later it will be crucial to use also paths which avoid final states. For this we introduce a special notation. First remove all arcs (p, a, q) where $q \in \mathcal{F}$ is a final state. Thus, final states do not have incoming arcs anymore in this reduced automaton. Let us write $p \overset{u}{\Longrightarrow} q$, if there is a path in this reduced automaton from state p to q which is labeled by the word u. Note that for such a path $p \overset{u}{\Longrightarrow} q$ we allow $p \in \mathcal{F}$, but on the path we never meet any final state again.

An NFA is called a deterministic finite automaton (DFA), if it has one initial state and for every state $p \in \mathcal{Q}$ and every letter $a \in \Sigma$ there is exactly one arc $(p, a, q) \in E$. In particular, a DFA in this paper is always complete, thus we can read every word to its end. We also write $p \cdot u = q$, if $p \xrightarrow{u} q$. This yields a (totally defined) function $\mathcal{Q} \times \Sigma^* \to \mathcal{Q}$, which defines an action of Σ^* on $\mathcal{Q}$ on the right.

In the following we need a DFA accepting L_1 as well as a DFA accepting L_2, but the DFA for L_2 has to work from right-to-left. Instead of introducing this concept we use a DFA (working as usual from left-to-right), which accepts $\overline{L_2}$.

This automaton has the same number of states (and is structurally isomorphic to) as a DFA accepting the *reversal language* of L_2.

As input we assume that the regular languages L_1 and $\overline{L_2}$ are specified by DFAs with state set $\mathcal{Q}_i$, state $q_{0i} \in \mathcal{Q}_i$ as initial state, and $\mathcal{F}_i \subseteq \mathcal{Q}_i$ as final states. We fix $n_i = |\mathcal{Q}_i|$ to be the number of states, $i = 1, 2$. By n we mean $\max\{n_1, n_2\}$. The input size is therefore the number n.

We also need the usual product DFA with state space

$$\mathcal{Q} = \{(p_1, p_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2 \mid \exists w \in \Sigma^* : (p_1, p_2) = (q_{01} \cdot w,\ q_{02} \cdot w)\}\,.$$

The action is given by $(p_1, p_2) \cdot a = (p_1 \cdot a,\ p_2 \cdot a)$. We let $n_{12} = |\mathcal{Q}|$. Hence, $n \le n_{12} \le n_1 \cdot n_2 \le n^2$, and $n = n_1 = n_{12}$ if $L_2 = \emptyset$ or $L_1 = \overline{L_2}$. In the following we work simultaneously in all three automata defined so far. Moreover, in $\mathcal{Q}_1$ and $\mathcal{Q}_2$ we have to work backwards. This leads to nondeterminism. Our first new construction concerns a special NFA in Section 3.1.

3 Main Results

The complexity results of this paper are the following:

Theorem 1. *The decision problem whether the hairpin completion $\mathcal{H}_k(L_1, L_2)$ is regular is* **NL***-complete.*

Theorem 2. *i.) The problem whether the hairpin completion $\mathcal{H}_k(L_1, L_2)$ is regular can be decided in time $\mathcal{O}(n^8)$.*
ii.) For $L_1 = \overline{L_2}$ it can be decided in time $\mathcal{O}(n^6)$.
iii.) For $L_2 = \emptyset$ it can be decided in time $\mathcal{O}(n^2)$.

An algorithm solving this problem is sketched in Section 3.3. For a proof of the strict time bounds and a proof of the **NL**-hardness we refer to [7].

The *growth* or *generating function* g_L of a formal language L is defined as: $g_L(z) = \sum_{m \ge 0} \left|L \cap \Sigma^{\le m}\right| z^m$. We can view g_L as a formal power series or as an analytic function in one complex variable where the radius of convergence is strictly positive. The radius of convergence is at least $1/|\Sigma|$.

It is well-known that the growth of a regular language L is effectively rational, i.e., a quotient of two polynomials. The same is true for unambiguous linear context-free languages. In particular, the growth is either polynomial or exponential. If the growth is exponential, then we find an algebraic number $\rho \in \mathbb{R}$ such that $\left|L \cap \Sigma^{\le m}\right|$ behaves essentially as ρ^m, see [3,10,2].

It was shown in [4] that $\mathcal{H}_k(L_1, L_2)$ is an linear context-free language. As a byproduct to our techniques to prove the complexity results above we find that $\mathcal{H}_k(L_1, L_2)$ is unambiguous, and hence its growth (i.e., generating function) is a rational function, see e.g. [15] for this well-known fact. We obtain:

Theorem 3. *The hairpin completion $\mathcal{H}_k(L_1, L_2)$ is an unambiguous linear context-free language with an effectively computable rational growth function.*

This result is proved in Section 3.2.

3.1 The NFA $\mathcal{A}$

In this section we define a certain NFA which is called simply $\mathcal{A}$. Almost all further results are done by exploring properties of this NFA. The NFA is a sort of product automaton over $\mathcal{Q} \times \mathcal{Q}_1 \times \mathcal{Q}_2 \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2 \times \mathcal{Q}_1 \times \mathcal{Q}_2$ where $\mathcal{Q}_1, \mathcal{Q}_2$ and $\mathcal{Q}$ are defined as in Section 2. The size of this automaton is $\mathcal{O}(n^4)$ in the worst case, and our decision algorithm will take into account all pairs of states in this NFA. Hence, $\mathcal{O}(n^8)$ might be an optimal time bound and the decision algorithm is not worse than quadratic in the size of the NFA $\mathcal{A}$.

For every quadruple $(p_1, p_2, q_1, q_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2 \times \mathcal{Q}_1 \times \mathcal{Q}_2$ we define a regular language $B(p_1, p_2, q_1, q_2)$ as follows:

$$B(p_1, p_2, q_1, q_2) = \{w \in \Sigma^* \mid p_1 \cdot w = q_1 \wedge p_2 \cdot \overline{w} = q_2\}.$$

We say that (p_1, p_2, q_1, q_2) is a *basic bridge* if $B(p_1, p_2, q_1, q_2) \neq \emptyset$. The idea behind of this notation is that $B(p_1, p_2, q_1, q_2)$ closes a gap between pairs (p_1, p_2) and (q_1, q_2) (which are on different sides). For a letter $a \in \Sigma$ we call (p_1, p_2, q_1, q_2) an *a-bridge* if $B(p_1, p_2, q_1, q_2) \cap a\Sigma^* \neq \emptyset$.

Lemma 1. *The number of basic bridges and a-bridges is bounded by* $\mathcal{O}(n_1^2 n_2^2)$. *A table containing all these bridges can be computed in time* $\mathcal{O}(n_1^2 n_2^2) \subseteq \mathcal{O}(n^4)$, *and there is a single-valued non-deterministic transduction working in* $\mathcal{O}(\log n)$ *space which outputs this table.*

Proof. To compute the basic bridges amounts to compute the transitive closure in some graph where the number of nodes and edges is in $\mathcal{O}(n_1 n_2)$. This gives the time bound. Once we have the bridges we can compute the a-bridges in time $\mathcal{O}(n_1^2 n_2^2)$.

If (p_1, p_2, q_1, q_2) is a basic bridge, we can verify this property in **NL**. Since **NL** is closed under complementation, we can output the whole table by a single-valued non-deterministic transduction in $\mathcal{O}(\log n)$ space.

We also need *levels* for $0 \leq \ell \leq k$, hence there are $k+1$ levels. By $[k]$ we denote in this paper the set $\{0, \ldots, k\}$. Define

$$\{((p_1, p_2), q_1, q_2, \ell) \in \mathcal{Q} \times \mathcal{Q}_1 \times \mathcal{Q}_2 \times [k] \mid (p_1, p_2, q_1, q_2) \text{ is a basic bridge}\}$$

as the state space of the NFA $\mathcal{A}$. Its size is bounded by $N \cdot (k+1) \in \mathcal{O}(N) \subseteq \mathcal{O}(n^4)$, where $N = n_{12} n_1 n_2$. We have $N = n^2$ for $L_2 = \emptyset$, and $N = n^3$ for $L_2 = \overline{L_1}$.

By a (slight) abuse of languages we call a state $((p_1, p_2), q_1, q_2, \ell)$ a *bridge*, and we keep in mind that there exists a word w such that $p_1 \cdot w = q_1$ and $p_2 \cdot \overline{w} = q_2$. Bridges are frequently denoted by (P, q_1, q_2, ℓ) with $P = (p_1, p_2) \in \mathcal{Q}$, $q_i \in Q_i$, $i = 1, 2$, and $\ell \in [k]$. Bridges are a central concept in the following.

The a-transitions in the NFA for $a \in \Sigma$ are given by the following arcs:

$$(P,\ q_1 \cdot \overline{a},\ q_2 \cdot \overline{a}, 0) \xrightarrow{a} (P \cdot a,\ q_1, q_2, 0) \qquad \text{for } q_i \cdot \overline{a} \notin \mathcal{F}_i,\ i = 1, 2,$$

$$(P,\ q_1 \cdot \overline{a},\ q_2 \cdot \overline{a}, 0) \xrightarrow{a} (P \cdot a,\ q_1, q_2, 1) \qquad \text{for } q_1 \cdot \overline{a} \in \mathcal{F}_1 \text{ or } q_2 \cdot \overline{a} \in \mathcal{F}_2,$$

$$(P,\ q_1 \cdot \overline{a},\ q_2 \cdot \overline{a}, \ell) \xrightarrow{a} (P \cdot a,\ q_1, q_2, \ell + 1) \qquad \text{for } 1 \leq \ell < k.$$

Observe that no state of the form $(P, q_1, q_2, 0)$ with $q_1 \in \mathcal{F}_1$ or $q_2 \in \mathcal{F}_2$ has an outgoing arc to level zero; we must switch to level one. There are no outgoing arcs on level k, and for each $(a, P, q_1, q_2, \ell) \in \Sigma \times \mathcal{Q} \times \mathcal{Q}_1 \times \mathcal{Q}_2 \times [k-1]$ there exists at most one arc $(P, q_1', q_2', \ell) \xrightarrow{a} (P \cdot a, q_1, q_2, \ell')$. Indeed, the triple (q_1', q_2', ℓ') is determined by (q_1, q_2, ℓ) and the letter a. Not all arcs exist because (P, q_1', q_2', ℓ) can be a bridge whereas $(P \cdot a, q_1, q_2, \ell')$ is not. Thus, there are at most $|\Sigma| \cdot N \cdot k \in \mathcal{O}(N)$ arcs in the NFA.

The set of initial states $\mathcal{I}$ contains all bridges of the form $(Q_0, q_1', q_2', 0)$ with $Q_0 = (q_{01}, q_{02})$. The set of final states $\mathcal{F}$ is given by all bridges (P, q_1, q_2, k) on level k.

Remark 1. The NFA $\mathcal{A}$ can be computed by Lemma 1 in time $\mathcal{O}(n_1^2 n_2^2)$ and by a single-valued non-deterministic transduction in $\mathcal{O}(\log n)$ space. Thus for both the polynomial time and the **NL** algorithm we can have direct access to $\mathcal{A}$ and we can assume that $\mathcal{A}$ is written on the input tape.

The next result shows the unambiguity of paths in the automaton $\mathcal{A}$.

Lemma 2. *Let $w \in \Sigma^*$ be the label of a path in $\mathcal{A}$ from a bridge $A = (P, p_1, p_2, \ell)$ to $A' = (P', p_1', p_2', \ell')$, then the path is unique. This means that $B = B'$ whenever $w = uv$ and*

$$A \xrightarrow{u} B \xrightarrow{v} A', \qquad\qquad A \xrightarrow{u} B' \xrightarrow{v} A'.$$

Proof. It is enough to consider $u = a \in \Sigma$. Let $B = (Q, q_1, q_2, m)$. Then we have $Q = P \cdot a$ and $q_i = p_i' \cdot \overline{v}$. If $\ell = 0$ and $p_i \notin \mathcal{F}_i$ for $i = 1, 2$, then $m = 0$, too. Otherwise $m = \ell + 1$. Thus, B is defined by A, A', and u, v. We conclude $B = B'$.

3.2 Structure Theorem and Rational Growth

For languages U and V we define the language V^U as follows:

$$V^U = \{uv\overline{u} \mid u \in U, v \in V\}.$$

Clearly, if U and V are regular, then V^U is linear context-free. We are interested in a disjoint union of languages V^U where for $w \in V^U$ the factorization $w = uv\overline{u}$ with $u \in U$ and $v \in V$ is unambiguous.

Theorem 4. *Let $T = \mathcal{I} \times \mathcal{F}$. For each $\tau = (I, F) \in T$ with $F = ((d_1, d_2), e_1, e_2, k)$ let R_τ be the (regular) set of words which label a path from the initial bridge I to the final bridge F and let $B_\tau = B(d_1, d_2, e_1, e_2)$. The hairpin completion is a disjoint union*

$$\mathcal{H}_k(L_1, L_2) = \bigcup_{\tau \in T} B_\tau^{R_\tau}.$$

Moreover, for each word in some $w \in B_\tau^{R_\tau}$ there is a unique factorization $w = \rho\beta\overline{\rho}$ with $\rho \in R_\tau$ and $\beta \in B_\tau$.

Corollary 1. *The hairpin completion $\mathcal{H}_k(L_1, L_2)$ is an unambiguous linear context-free language and it has a rational growth function. The growth can be directly calculated by the growth of the regular languages R_τ and B_τ.*

Corollary 1 allows to compare the growth of L_1 and L_2 with the growth of their hairpin completion $\mathcal{H}_k(L_1, L_2)$. It is also a slightly more precise version of Theorem 3.

3.3 Complexity for Testing the Regularity of $\mathcal{H}_k(L_1, L_2)$

First Test. The automaton $\mathcal{A}$ accepts the union of the languages R_τ as defined in Theorem 4. If the accepted language is finite then all R_τ are finite and hence all $B_\tau^{R_\tau}$ are regular. This leads to the following result:

Proposition 1. *i.) If the accepted language of the NFA $\mathcal{A}$ is finite, then the hairpin completion $\mathcal{H}_k(L_1, L_2)$ is regular.*
ii.) If L_1 or L_2 is finite, but the accepted language of $\mathcal{A}$ is infinite, then the hairpin completion $\mathcal{H}_k(L_1, L_2)$ is not regular.

Test 1: Check either by some **NL**-algorithm or in time $\mathcal{O}(N) \subseteq \mathcal{O}(n^4)$ (in time $\mathcal{O}(n^2)$, if L_1 or L_2 is empty) whether the accepted language of the NFA $\mathcal{A}$ is finite.

If "yes" ($=L(\mathcal{A})$ is finite), then output that $\mathcal{H}_k(L_1, L_2)$ is regular. If "no", but L_1 or L_2 is finite, then output that $\mathcal{H}_k(L_1, L_2)$ is not regular.

Second Test. From now on we may assume that the automaton $\mathcal{A}$ accepts an infinite language and both L_1 and L_2 are infinite as well. We assume that all states are reachable from initial bridges and lead to some final bridges. (Recall that graph reachability can be checked in **NL**.)

Let K be the set of non-trivial strongly connected components of the automaton $\mathcal{A}$ (read as a directed graph). For $\kappa \in K$ let $N_\kappa = |\kappa|$ the number of states in the component κ. Let us choose some $A_\kappa \in \kappa$ and some shortest non-empty word $v_\kappa \in \Sigma^+$ such that there is a path in $\mathcal{A}$ labeled by v_κ from A_κ to A_κ.

The next lemma tells us that for a regular hairpin completion $\mathcal{H}_k(L_1, L_2)$ the word v_κ is uniquely defined by A_κ, its length is N_κ, and its conjugacy class depends only on κ.

Lemma 3. *Assume that the hairpin completion $\mathcal{H}_k(L_1, L_2)$ is regular.*
1.) Let $A_\kappa \xrightarrow{v_\kappa} A_\kappa$ as above and $A_\kappa \xrightarrow{w} C$ be a path in $\mathcal{A}$ to some final bridge. Then the word w is a prefix of some word in v_κ^+.
2.) The word v_κ and the loop $A_\kappa \xrightarrow{v_\kappa} A_\kappa$ are uniquely defined by the state A_κ and we have $|v_\kappa| = N_\kappa$.
3.) The loop $A_\kappa \xrightarrow{v_\kappa} A_\kappa$ visits every other state $B \in \kappa$ exactly once. Thus, the loop defines an Hamiltonian cycle of κ.

Remark 2. We decompose the automaton $\mathcal{A}$ in its strongly connected components by the algorithm of Tarjan in time $\mathcal{O}(N)$. (Note that we have $K \neq \emptyset$

since $|L(\mathcal{A})|$ is infinite.) This is also possible by some single-valued non-deterministic transduction. Putting some linear order on the set of bridges, we can assign to each $\kappa \in K$ the least $A_\kappa \in \kappa$. If $\mathcal{H}_k(L_1, L_2)$ is regular, then (by Lemma 3) we can output the uniquely defined words v_κ for all $\kappa \in K$. We observe that $\sum_{\kappa\in K} |v_\kappa| = \sum_{\kappa\in K} N_\kappa \leq N$. So, the list of all v_κ is computable in time $\mathcal{O}(N)$ and also by some single-valued non-deterministic transduction, in case $\mathcal{H}_k(L_1, L_2)$ is regular.

Test 2: It has two parts. Part I: For each strongly connected component $\kappa \in K$ compute a shortest word v with $0 < |v| \leq N_\kappa$ such that $A_\kappa \xrightarrow{v} A_\kappa$ is a loop in the automaton $\mathcal{A}$. If $|v| \neq N_\kappa$, then **stop** and output that $\mathcal{H}_k(L_1, L_2)$ is not regular. Part II: If $|v| = N_\kappa$ for all κ, then let L_κ be the accepted language of $\mathcal{A}$ when the bridge A_κ is used as initial state. Let $\mathrm{Pref}(v^+)$ be the language of prefixes of words in v^+. (Note that a DFA for the complement of $\mathrm{Pref}(v^+)$ has $N_\kappa + 1$ states.) If we do not find $L_\kappa \subseteq \mathrm{Pref}(v^+)$, then **stop** and output that $\mathcal{H}_k(L_1, L_2)$ is not regular.

Part I can be done in time $\mathcal{O}(\sum_{\kappa\in K} N_\kappa) \subseteq \mathcal{O}(N)$ or in **NL**. Part II can be done in time $\mathcal{O}(\sum_{\kappa\in K} N_\kappa \cdot N) \subseteq \mathcal{O}(N^2) \subseteq \mathcal{O}(n^8)$. The **NL**-algorithm for Part II is based on the fact that we can guess a position m where the m-th letter of $w \in L_\kappa$ differs from the $(m \bmod N_\kappa)$-th letter of v_κ.

Remark 3. Henceforth we may assume that Test 2 was successful and following Remark 2 we assume that the list of all words v_κ is available. Thus, we can think that the list $(v_\kappa; \kappa \in K)$ is written on the input tape. For the **NL**-algorithm we perform another single-valued non-deterministic transduction to achieve this.

Third and Fourth Test. We fix a strongly connected component $\kappa \in K$ of $\mathcal{A}$. We let $A = A_\kappa = ((p_1, p_2), q_1, q_2, 0)$ and $v = v_\kappa$ as above. By u we denote some word leading from an initial bridge $((q_{01}, q_{02}), q_1', q_2', 0)$ to A. (The following tests do not rely on the choice of u.) The main idea is to investigate runs through the DFAs for L_1 and $\overline{L_2}$ where $s, t \geq n$.

$$L_1 : \quad q_{01} \xrightarrow{u} p_1 \xrightarrow{v^s} p_1 \xrightarrow{xy} c_1 \xrightarrow{z} d_1 \xrightarrow{\overline{x}} e_1 \xRightarrow{\overline{v}^{n_1}} q_1 \xRightarrow{\overline{v}^*} q_1 \xRightarrow{\overline{u}} q_1'$$

$$\overline{L_2} : \quad q_{02} \xrightarrow{u} p_2 \xrightarrow{v^t} p_2 \xrightarrow{x} c_2 \xrightarrow{\overline{z}} d_2 \xrightarrow{\overline{y}\overline{x}} e_2 \xRightarrow{\overline{v}^{n_2}} q_2 \xRightarrow{\overline{v}^*} q_2 \xRightarrow{\overline{u}} q_2'$$

We investigate the case where $uv^s xyz\overline{x}\overline{v}^t\overline{u} \in \mathcal{H}_k(L_1, L_2)$ for all $s \geq t$ and where (by symmetry) this property is due to the longest prefix belonging to L_1.

The following lemma is the most technical one in our paper.

Lemma 4. *Let $x, y, z \in \Sigma^*$ be words and $(d_1, d_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$ with the following properties:*

1.) $k \leq |x| < |v| + k$ and x is a prefix of some word in v^+.
2.) $0 \leq |y| < |v|$ and xy is the longest common prefix of xyz and some word in v^+.
3.) $z \in B(c_1, c_2, d_1, d_2)$, where $c_1 = p_1 \cdot xy$ and $c_2 = p_2 \cdot x$.

4.) $q_1 = d_1 \cdot \overline{x}\overline{v}^{n_1}$ *and during the computation of* $d_1 \cdot \overline{x}\overline{v}^{n_1}$ *we see after exactly* k *steps a final state in* $\mathcal{F}_1$ *and then never again.*

5.) $q_2 = d_2 \cdot \overline{y}\overline{x}\overline{v}^{n_2}$ *and, let* $e_2 = d_2 \cdot \overline{y}\overline{x}$*, during the computation of* $e_2 \cdot \overline{v}^{n_2}$ *we do not see a final state in* $\mathcal{F}_2$*.*

If $\mathcal{H}_k(L_1, L_2)$ *is regular, then* $xyz\overline{x}\overline{v} = \mu\delta\beta\overline{\delta}\overline{\mu}$ *where* $|\delta| = k$ *and* $\delta\beta\overline{\delta}\overline{\mu}\overline{u} \in L_2$*.*

Lemma 5. *The existence of words* $x, y, z \in \Sigma^*$ *and states* $(d_1, d_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$ *satisfying 1.) to 5.) of Lemma 4, but where for all factorizations* $xyz\overline{x}\overline{v} = \mu\delta\beta\overline{\delta}\overline{\mu}$ *we have* $p_2 \cdot \mu\delta\overline{\beta}\overline{\delta} \notin \mathcal{F}_2$ *(and accordingly* $\delta\beta\overline{\delta}\overline{\mu}\overline{u} \notin L_2$*), can be decided in time* $\mathcal{O}(n_{12}^2 n_1^2 n_2^2) \subseteq \mathcal{O}(n^8)$ *and in* **NL***.*

Proof. It is enough to perform Tests 3, 4 below and to prove the complexity.

The tests distinguish whether the word z is non-empty or empty.

Test 3: Decide the existence of words $x, y, z \in \Sigma^*$ with $z \neq 1$ and states $(d_1, d_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$ satisfying 1.) to 5.) of Lemma 4, but where for all factorizations $xyz\overline{x}\overline{v} = \mu\delta\beta\overline{\delta}\overline{\mu}$ we have $p_2 \cdot \mu\delta\overline{\beta}\overline{\delta} \notin \mathcal{F}_2$. If we find such a situation, then **stop** and output that $\mathcal{H}_k(L_1, L_2)$ is not regular.

Test 4: Decide the existence of words $x, y \in \Sigma^*$ and states $(d_1, d_2) \in \mathcal{Q}_1 \times \mathcal{Q}_2$ satisfying 1.) to 5.) of Lemma 4 with $z = 1$, but where for all factorizations $xy\overline{x}\overline{v} = \mu\delta\beta\overline{\delta}\overline{\mu}$ we have $p_2 \cdot \mu\delta\overline{\beta}\overline{\delta} \notin \mathcal{F}_2$. If we find such a situation, then **stop** and output that $\mathcal{H}_k(L_1, L_2)$ is not regular.

The correctness of both tests follows by Lemma 4, but even termination of Test 3 is not completely obvious. Termination is due to condition that xy is the longest common prefix of xyz and some word in v^+. This means, if $z \neq 1$, then there exists a letter a such that $z \in a\Sigma^*$ and xya is no prefix of any word in v^+. Now $|y| < |v|$, hence we see that $xyz\overline{x}\overline{v} = \mu\delta\beta\overline{\delta}\overline{\mu}$ implies $\mu\delta \leq xy$.

Thus it is enough to check the computation starting in state $d_2 \in \mathcal{Q}_2$ when reading the word $\overline{y}\overline{x}$. Test 3 is successful if we find such a computation which after more than $k-1$ steps does not meet any final state in $\mathcal{F}_2$. We do not need the word z, we just have to know that (c_1, c_2, d_1, d_2) is in the precomputed table of a-bridges (cf. Lemma 1) where a is a letter such that xya is no prefix of any word in v^+. It is obvious that Test 3 can be performed in polynomial time as well as in **NL**.

Test 4 is for $z = 1$, so in any case the number of factorizations $xy\overline{x}\overline{v} = \mu\delta\beta\overline{\delta}\overline{\mu}$ is polynomial. It is again obvious that Test 4 can be performed polynomial time as well as in **NL**.

The following lemma completes the proof of Theorem 1 and 2.

Lemma 6. *Suppose no outcome of Tests 1, 2, 3, and 4 is "not regular". Then the hairpin completion* $\mathcal{H}_k(L_1, L_2)$ *is regular.*

Acknowledgement

We thank the anonymous referees for many useful remarks and hints.

References

1. Baker, B.S., Book, R.V.: Reversal-bounded multi-pushdown machines. In: Annual IEEE Symposium on Foundations of Computer Science, pp. 207–211 (1972)
2. Berstel, J., Reutenauer, C.: Rational series and their languages. Springer, New York (1988)
3. Ceccherini-Silberstein, T.: On the growth of linear languages. Advances in Applied Mathematics 35(3), 243–253 (2005)
4. Cheptea, D., Martin-Vide, C., Mitrana, V.: A new operation on words suggested by DNA biochemistry: Hairpin completion. Transgressive Computing, 216–228 (2006)
5. Deaton, R., Murphy, R., Garzon, M., Franceschetti, D., Stevens, S.: Good encodings for DNA-based solutions to combinatorial problems. In: Proc. of DNA-Based computers DIMACS Series, vol. 44, pp. 247–258 (1998)
6. Diekert, V., Kopecki, S., Mitrana, V.: On the hairpin completion of regular languages. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 170–184. Springer, Heidelberg (2009)
7. Diekert, V., Kopecki, S.: Complexity Result and the Growths of Hairpin Completions of Regular Languages. Technical Report Computer Science 2010/04, University of Stuttgart (June 2010)
8. Garzon, M., Deaton, R., Neathery, P., Murphy, R., Franceschetti, D., Stevens, E.: On the encoding problem for DNA computing. In: The Third DIMACS Workshop on DNA-Based Computing, pp. 230–237 (1997)
9. Garzon, M., Deaton, R., Nino, L., Stevens Jr., S., Wittner, M.: Genome encoding for DNA computing. In: Proc. Third Genetic Programming Conference, pp. 684–690 (1998)
10. Gawrychowski, P., Krieger, D., Rampersad, N., Shallit, J.: Finding the growth rate of a regular or context-free language in polynomial time. In: Ito, M., Toyama, M. (eds.) DLT 2008. LNCS, vol. 5257, pp. 339–358. Springer, Heidelberg (2008)
11. Greibach, S.A.: A note on undecidable properties of formal languages. Mathematical Systems Theory 2(1), 1–6 (1968)
12. Hopcroft, J.E., Ulman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
13. Kari, L., Konstantinidis, S., Losseva, E., Sosík, P., Thierrin, G.: Hairpin structures in DNA words. In: Carbone, A., Pierce, N.A. (eds.) DNA 2005. LNCS, vol. 3892, pp. 158–170. Springer, Heidelberg (2006)
14. Kari, L., Mahalingam, K., Thierrin, G.: The syntactic monoid of hairpin-free languages. Acta Inf. 44(3-4), 153–166 (2007)
15. Kuich, W.: On the entropy of context-free languages. Information and Control 16, 173–200 (1970)
16. Manea, F., Mitrana, V., Yokomori, T.: Two complementary operations inspired by the DNA hairpin formation: Completion and reduction. Theor. Comput. Sci. 410(4-5), 417–425 (2009)
17. Papadimitriou, C.H.: Computatational Complexity. Addison Wesley, Reading (1994)
18. Sakamoto, K., Gouzu, H., Komiya, K., Kiga, D., Yokoyama, S., Yokomori, T., Hagiya, M.: Molecular Computation by DNA Hairpin Formation. Science 288(5469), 1223–1226 (2000)

On Straight Words and Minimal Permutators in Finite Transformation Semigroups*

Attila Egri-Nagy and Chrystopher L. Nehaniv

Royal Society Wolfson BioComputation Research Lab
Centre for Computer Science & Informatics Research, University of Hertfordshire
Hatfield, Hertfordshire AL10 9AB, United Kingdom
{A.Egri-Nagy,C.L.Nehaniv}@herts.ac.uk

Abstract. Motivated by issues arising in computer science, we investigate the loop-free paths from the identity transformation and corresponding straight words in the Cayley graph of a finite transformation semigroup with a fixed generator set. Of special interest are words that permute a given subset of the state set. Certain such words, called minimal permutators, are shown to comprise a code, and the straight ones comprise a finite code. Thus, words that permute a given subset are uniquely factorizable as products of the subset's minimal permutators, and these can be further reduced to straight minimal permutators. This leads to insight into structure of local pools of reversibility in transformation semigroups in terms of the set of words permuting a given subset. These findings can be exploited in practical calculations for hierarchical decompositions of finite automata. As an example we consider groups arising in biological systems.

1 Introduction

From the computational perspective it is very important to know how a particular element of a transformation semigroup can (efficiently) be generated. Of special interest are elements of the semigroup that permute a subset of the state set, as the hierarchical decomposition of the semigroup [8] depends on the group components [1,2]. Here we study the ways in which a particular transformation can be expressed without any redundancy. These generator words head towards the target transformation without without revisiting any transformation along the way, so they are called *straight*. Straight words also encode the information describing all possible ways that particular semigroup element can be generated.

Notation. For a finite transformation semigroup (X, S) we fix a generator set of transformations $T = \{t_1, \ldots, t_n\}$, so $S = \langle T \rangle$. We also consider the generators as symbols, thus a finite product of the generator elements becomes a word in T^+ (the free semigroup on generators T whose associative binary operation is

* Partial support for this work by the OPAALS EU project FP6-034824 is gratefully acknowledged.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 115–124, 2011.

concatenation). It is then convenient to consider the empty word ϵ as the identity map. We need to distinguish between the word (often thought of as a sequence of input symbols) and the transformation it realizes: for the word we just write the generator symbols in sequence $t_{i_1} \dots t_{i_n} \in T^+$ while the transformation is denoted by $\overrightarrow{t_{i_1} \dots t_{i_n}} \in S$, where the arrow indicates the order in which the generator elements are multiplied and emphasizes that is a mapping.

For transformations, we either use the usual 2-line notation for mappings, or if it would become too space consuming we apply the linear notation suggested in [5]. This is a natural extension of the cyclic notation of permutations. Considering the mappings as digraphs, each transformation consists of one or more components. Each component contains a cycle (possibly a trivial cycle). Unlike the permutation case, the points in the cycle can have incoming edges, denoted by $[\ \text{source}_1, \dots, \text{source}_m\ ;\ \text{target}\ ]$ where target is the point in the cycle. If a source point also has incoming edges from other points the same square bracket structure is applied again recursively. We can say that the points in the cycle are sinks of trees. Parentheses indicate the existence of a nontrivial permutation of the sink elements of the trees, but not of their sources: $\big([\ \text{sources}_1;\ \text{target}_1\ ], \dots, [\ \text{sources}_k;\ \text{target}_k\ ]\big)$. This corresponds to the cycle $(\ \text{target}_1, \dots,\ \text{target}_k)$ but at the same time it contains information on transient states. The order is arbitrary if there are more than one component. (See below for examples.)[1]

2 Straight Words

If the goal is to generate a transformation $s \in S$ as quickly as possible without any digression, then in each step of the generation a new transformation should appear. Also, if a prefix generates the identity map, so strictly speaking we did nothing so far, then the prefix can be discarded. More precisely,

Definition 1 (Straight Words). *Let $s \in S$ be a transformation generated by the word $t_{i_1} \dots t_{i_m} \in T^+$, so $s = \overrightarrow{t_{i_1} \dots t_{i_m}}$, then this word is* straight *if*

$$\overrightarrow{t_{i_1} \dots t_{i_k}} \neq \epsilon,\ k \in \{1, \dots, m-1\} \tag{1}$$

$$\overrightarrow{t_{i_1} \dots t_{i_k}} = \overrightarrow{t_{i_1} \dots t_{i_l}} \Rightarrow k = l \quad (1 \leq k, l \leq m). \tag{2}$$

Example 1 (Cyclic (monogenic) semigroup). Let $X = \{1, 2, 3, 4\}$ and $t = \left(\begin{smallmatrix}1&2&3&4\\2&4&1&2\end{smallmatrix}\right)$, or, in the alternative notation, $t = ([[3; 1]; 2], 4)$. The semigroup generated by t is

$$\langle t \rangle = \{t = \left(\begin{smallmatrix}1&2&3&4\\2&4&1&2\end{smallmatrix}\right), t^2 = \left(\begin{smallmatrix}1&2&3&4\\4&2&2&4\end{smallmatrix}\right), t^3 = \left(\begin{smallmatrix}1&2&3&4\\2&4&4&2\end{smallmatrix}\right)\}.$$

t, t^2 and t^3 are straight words, but these are the only ones. Higher powers, like $t^4 = t^2$, already repeatedly visit transformations. This example shows that being straight is not necessarily connected to the formal notion of containing repeated subwords.

[1] Our notation is slightly different from [5] as we do not use square brackets for a singleton source.

Example 2 (Cyclic group). Let $g = (1, 2, 3)$ be a permutation, then $g^3 = \epsilon$ is a straight word producing the identity map. This example justifies condition 1 in Definition 1, as we allow the identity transformation at the end of a word, but not inside.

Definition 2 (Trajectory). *Let $s_1, \ldots, s_n$ be a sequence of semigroup elements, $s_j \in S$. Then the sequence is a* trajectory *if for all $s_j, 1 \leq j < n$ there is a generator $t_i \in T$ such that $s_j \cdot t_i = s_{j+1}$.*

A trajectory is a path in the Cayley graph of the semigroup starting at the trivial transformation. We can associate a trajectory with a word.

Definition 3 (Trajectory of a word). *Given a word $t_{i_1} \ldots t_{i_m}$, its trajectory is calculated by taking the products of prefixes: $\epsilon, \overrightarrow{t_{i_1}}, \overrightarrow{t_{i_1} t_{i_2}}, \ldots, \overrightarrow{t_{i_1} \ldots t_{i_m}}$.*

Now we can give an alternative definition of straight words.

Alternative Definition (Straight Words). *A word is* straight *if all the elements of its trajectory are distinct, except the case of loops when the first and the last element coincide (and equal ϵ).*

Straight Words and Transformations. From finiteness it follows that the straight words cannot be extended beyond some finite length, since there are finitely many elements of the semigroup and each prefix should realize a distinct semigroup element. An obvious bound on the length of the straight words is $|S|$. This bound is reached in Example 2. We also observe that all semigroup elements can be realized by a straight word.

Lemma 1. *Let (X, S) be a transformation semigroup with states X and semigroup S generated by T. Each semigroup element $s \in S$ can be realized by a straight word in the letters of T.*

Proof. Let $s = \overrightarrow{t_{i_1} \ldots t_{i_m}}$. If $t_{i_1} \ldots t_{i_m}$ is not straight then there is $k \neq l$ such that $\overrightarrow{t_{i_1} \ldots t_{i_k}} = \overrightarrow{t_{i_1} \ldots t_{i_l}}$. Suppose that $k < l$. Then the product $\overrightarrow{t_{i_1} \ldots t_{i_k} t_{i_{l+1}} \ldots t_{i_m}}$ still generates s, after we cut out $t_{i_{k+1}} \ldots t_{i_l}$.

Similarly, in case an identity appears at some position (not the final one) in a trajectory then the whole prefix can be ignored up to that point. If the reduced word is not straight then we can repeat either processes. Due to finiteness this method will stop, and thus produce a straight word generating s. □

Another way to see that there is at least one straight word for each transformation is to observe that the first occurrences of transformations in a breadth-first generation of S by T are produced by straight words.

Corollary 1. *Any minimal length word generating $s \in S$ is a straight word.*

We have seen that for each semigroup element we can give at least one straight generator word. The following example shows that there can be more straight words for a mapping.

Example 3 (Constant Maps). Let $t_1 = \left(\begin{smallmatrix}1&2\\1&1\end{smallmatrix}\right)$ and $t_2 = \left(\begin{smallmatrix}1&2\\2&2\end{smallmatrix}\right)$ be two generators, then t_1 and t_2t_1 are each straight words for $\overrightarrow{t_1}$, while t_2 and t_1t_2 are straight and both realize $\overrightarrow{t_2}$. Constant maps render the transformations before them negligible.

Synonym Straight Words. Different straight words may represent the same transformation. For example, if we add a second generator, $r = \left(\begin{smallmatrix}1&2&3&4\\4&2&2&4\end{smallmatrix}\right)$ to Example 1, then clearly r and t^2 are words with this property. Moreover, two different words may have the same trajectories.

Generalization: Straight Paths. We can study straight words in a more general settings, we look for straight words $w = t_{i_1} \dots t_{i_m}$ such that $s \cdot \overrightarrow{w} = r$, where $r, s \in S$. Actually these arise as labels of 'straight paths' in the Cayley graph of the semigroup between nodes s and r, i.e. simple paths that do not cross themselves but go directly from s to r. We get the special case of straight words when $s = \epsilon$.

Computational Implementation. Computational enumeration of straight words can easily be done with a backtrack algorithm. We implemented the search algorithm in the `SgpDec` software package [4] in the `GAP` computer algebra system [6].

3 Minimal Straight Words and Permutations of Subsets

From now on we focus on straight words that induce permutations on a subset of the state set. The *full permutator semigroup* $\mathrm{Perm}(Y)$ for a subset $Y \subseteq X$ in (X, S) is

$$Perm(Y) = \{s \in S : Y \cdot s = Y\}.$$

Elements of $Perm(Y)$ are called also *permutators of* Y. $Perm(Y)$ is closed under products, so by finiteness it restricts to a group of permutations acting on Y. The restrictions of elements of $Perm(Y)$ to Y thus comprise a permutation group or 'pool of reversibility' or 'natural subsystem' within the transformation semigroup (X, S). However, while any $s \in \mathrm{Perm}(Y)$ is also defined on all of X it is not generally a permutation of X. Elements of $Perm(Y)$ may agree on Y but disagree on $X \setminus Y$, so $Perm(Y)$ may not itself be a group nor act faithfully on Y. We also call a word a *permutator word* if it realizes a permutator transformation.

Example 4 (Cyclic uniquelly labelled digraph as an automaton). The generator set consists of 3 elementary collapsings, $T = \{a = 1 \mapsto 2,\ b = 2 \mapsto 3,\ c = 3 \mapsto 1\}$. The generated semigroup has 21 elements and, in the notation introduced above, the straight words of the semigroup elements are: $[3;1] = c$, $[2;3] = b$, $[1;2] = a$, $[[2;3];1] = cb$, $[[1;2];3] = ba$, $[[3;1];2] = ac$, $([1;2],3) = cba$, $([3;1],2) = bac$, $(1,[2;3]) = acb$, $(1,[3;2]) = cbac$, $([2;1],3) = bacb$, $([1;3],2) = acba$, $[[2;1];3] = cbacb$, $[[1;3];2] = bacba$, $[[3;2];1] = acbac$, $[1;3] = cbacba$,

$[3;2] = bacbac$, $[2;1] = acbacb$, plus the constant maps that are represented by a lot more straight words:

$[1,3;2]$: $abca, aca, acbabca, acbaca, acbacbca, acbca, babca, baca, bacbabca, bacbaca, bacbca, bca, ca, cbabca, cbaca, cbacbabca, cbacbca, cbca$
$[2,3;1]$: $abc, acabc, acbabc, acbacabc, acbacbc, acbc, babc, bacabc, bacbabc, bacbacabc, bacbc, bc, cabc, cbabc, cbacabc, cbacbabc, cbacbc, cbc$
$[1,2;3]$: $ab, acab, acbab, acbacab, acbacbcab, acbcab, bab, bacab, bacbab, bacbacab, bacbcab, bcab, cab, cbab, cbacab, cbacbab, cbacbcab, cbcab$.

Now let $Y = \{1,2\}$. There are exactly 4 straight words permuting Y, bac and $cbac$ realizing the transposition $(1,2)$, and c and $bacbac$ realizing the identity, thus $\{\overrightarrow{c}, \overrightarrow{bac}, \overrightarrow{cbac}, \overrightarrow{bacbac}\} = \mathrm{Perm}(Y)$. These words happen to give 4 distinct transformations of $\{1,2,3\}$. Note that two of these words are products of two of the others. There are also many permutator words that not straight, e.g. $\overrightarrow{baac} = \overrightarrow{bbac} = ([3;1],2)$, which is $\overrightarrow{bac}$.

A word w is a *minimal permutator* of Y if w represents an element of $\mathrm{Perm}(Y)$ and w is not a product of two or more words permuting Y. That is, $w \neq w_1 w_2$ for any words w_1 and w_2 representing elements of $\mathrm{Perm}(Y)$. The set of minimal permutators is not necessarily finite, as we can use idempotents to "pump in the middle" like $ba^n c$ in Example 4. Therefore we turn our attention to the set of *straight, minimal permutator words* of Y, denoted by $M_S(Y)$.

Fact 1. *The set $M_S(Y)$ of straight minimal permutator words for Y is finite.*

Proof. The assertion easily follows from the fact that straight words are bounded in length. □

Now we need to show that we do not lose anything by discarding the words that are not straight, i.e. we can still generate the full permutator semigroup. We will use the following obvious fact.

Fact 2. *If $w = uv$ permutes Y and u permutes Y, then v permutes Y.*

Theorem 1. *In the free semigroup T^+ on the generators of S, the minimal permutator words $M(Y)$ of Y generate the subsemigroup of all words realizing elements of* $\mathrm{Perm}(Y)$. *That is,*

$$\langle M(Y) \rangle = \textit{all words representing elements of } \mathrm{Perm}(Y).$$

Moreover, the straight minimal permutators $M_S(Y)$ of Y generate a subsemigroup of words realizing all elements of $Perm(Y)$.

Proof. Let $p = t_1 \dots t_k$ represent an element of $\mathrm{Perm}(Y)$. We show p is a product of minimal permutators by induction on k. Either p is a minimal permutator or there is a least j strictly less than k so that $t_1 \dots t_j$ permutes Y. Now $t_1 \dots t_j$ is a minimal permutator of Y and $p = (t_1 \dots t_j)(t_{j+1} \dots t_k)$ with each of the expressions in parentheses permuting Y. The length of the second word is strictly less than k, so by induction hypothesis, it too can be written as a product of

minimal permutators of Y. This proves that an arbitrary word p representing an element of $\mathrm{Perm}(Y)$ can be factored as a product of minimal permutators of Y. Each minimal permutator factor can be shortened by removing letters if necessary to a straight word. The result follows. □

Theorem 2. *Any word w representing a permutator of Y can be factored uniquely into a product of minimal permutators of Y.*

Proof. By the previous theorem, we can write $w = w_1 \cdots w_k$, where each w_i is a minimal permutator word of Y. Suppose w can also be written as $w = w'_1 \dots w'_\ell$, where again each w'_i represents is a minimal permutator word for Y. We show $\ell = k$ and $w_j = w'_j$ for all j $(1 \leq j \leq \ell)$. If this were not the case, then let i be the least index such that $w_i \neq w'_i$. Without loss of generality, assume $|w_i| \leq |w'_i|$. It follows then that $w'_i = w_i v$ for some nonempty word v. By Fact 2, v represents a permutator of Y. But we have then written w'_i as a product of permutator words, this contradicts the choice of w'_i as a minimal permutator. It follows $w_i = w'_i$ for all i, and, since the two factorizations are of the same word, that $\ell = k$. □

Corollary 2. *The minimal permutator words are a code.*

Corollary 3. *The straight minimal permutator words are a finite code.*

The last corollary shows the usefulness of straight words, when looking for permutators instead of an infinite search space we can restrict the search to a finite set of words.

Fact 3. *For a minimal permutator word w, there is a (in general non-unique) straight minimal permutator word $red(w)$ obtained from w by removing some letters such that $\overrightarrow{w} = \overrightarrow{red(w)}$.*

Proof. Considering the Cayley graph of the transformation semigroup (X, S) with generators T. This has vertices $S^1 = S \cup \{\epsilon\}$, where ϵ denotes the identity mapping on X, and edges $s \xrightarrow{t} s'$, where $s' = s\,\overrightarrow{t}$ with $t \in T$, $s, s' \in S^1$. Now, by the alternative definition of straight words, it is clear that a word is straight if and only if the path it labels starting at ϵ and has no loop (does not enter any node more than once). Noting that adding or removing loops to the path corresponding to a product does not change its endpoint, we conclude that removing contiguous subwords from the word w corresponding to loops, iteratively if necessary, results in a path with no loops (or a simple loop at ϵ), corresponding to a straight word w' representing the same transformation as w. □

Theorem 3. *There is a well-defined homomorphism $\phi : M(Y)^+ \twoheadrightarrow M_S(Y)^+$ from the semigroup of permutator words onto the semigroup generated by minimal straight permutator words, where for each minimal permutator $w \in M(Y)$, $\phi(w)$ is a straight word having the same trajectory as w except for the removal of loops. Furthermore, ϕ is a retraction, i.e. $\phi(w) = w$ for all words w in $M_S(Y)$ (and hence is the identity on $M_S(Y)^+$). For all permutator words $w \in M(Y)^+$, w and $\phi(w)$ act by the same permutation of Y, and moreover by the same mapping on X.*

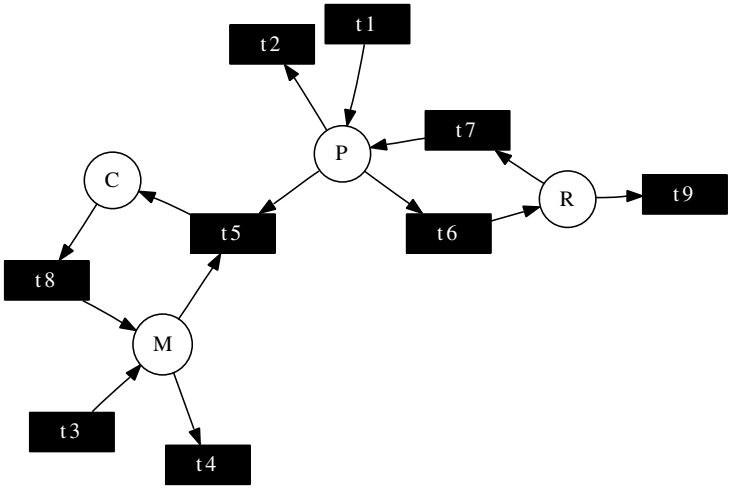


Fig. 1. Petri net for the p53-Mdm2 regulatory pathway. P = p53, M = Mdm2, C = p53-Mdm2, R = p53*.

Proof. To get a well-defined homomorphism from the minimal permutator code to the straight word minimal permutator code, one only needs to choose some reduction for each minimal permutator (any reduction at all would work). The reason why one gets a homomorphism is due to that fact the minimal permutators are a code, hence free generators of a free semigroup, so we need only say where each generator goes and extend uniquely by freeness. □

The reduction of a minimal permutator to a straight word need not be unique. This comes from the fact that synonym straight words do exist. Thus the homomorphism of the theorem need not be unique. One natural way to choose the reduction $red(w)$ is the following: given a minimal permutator word w that is not straight, find the first node (along its trajectory) that is later repeated. Start deleting letters after the letter that first takes us into this node. Find the last time this node occurs. Delete all letters from there up to and including the one taking us into the node for the last time. This process removes at least one letter since the word was not straight. Repeat the procedure until the resulting word is straight. This necessarily terminates with a reduced form for w, realizing the same transformation by a straight word obtained from w by excising some subwords ('removing loops' in the trajectory as described).

4 A Biological Example

It seems that in constructing interesting examples the human mind is somewhat constrained and reverts back to special cases. Therefore studying "naturally occurring" transformation semigroups can be useful, so here we investigate a

biological example. We should also mention that in exchange semigroup and automata theory can also provide useful tools for other sciences [9].

The p53-Mdm2 Regulatory Pathway. Biological networks are frequently modelled by Petri nets and thus it is not difficult to convert such a model to a transformation semigroup [3]. Figure 1 shows such a model of the p53-Mdm2 regulatory pathway, which is important in the cellular response to ionizing radiation and can trigger self-repair or, in extreme cases, the onset of programmed cell-death (apoptosis). This pathway is involved in ameliorating DNA damage and preventing cancer [7]. Figure 2 shows the corresponding finite automata with 16 states in which two levels of each of the 4 molecular species involved are distinguished. Corresponding to the transitions we have the following generator transformations:

$$
\begin{aligned}
t_1 &= [1;2][3;4][5;6][7;9][8;10][11;12][13;14][15;16]\\
t_2 &= [2;1][4;3][6;5][9;7][10;8][12;11][14;13][16;15]\\
t_3 &= [1;3][2;4][5;7][6;9][8;11][10;12][13;15][14;16]\\
t_4 &= [3;1][4;2][7;5][9;6][11;8][12;10][15;13][16;14]\\
t_5 &= [4,12;8][9,16;13]\\
t_6 &= [2,6;5][4,9;7][10,14;13][12,16;15]\\
t_7 &= [5,6;2][7,9;4][13,14;10][15,16;12]\\
t_8 &= [8,11;3][10,12;4][13,15;7][14,16;9]\\
t_9 &= [5;1][6;2][7;3][9;4][13;8][14;10][15;11][16;12]
\end{aligned}
$$

Analysis of a Permutator Subsemigroup. None of the above generators contain a cycle, so the existence of a nontrivial permutation group cannot be simply read off. The generated semigroup has 316,665 elements. The decomposition of the semigroup shows that it has (several copies of) the following group components: cyclic group C_2 acting on 4, symmetric group S_3 acting on 3 and C_2 acting on 2 states.

We pick the set $\{3,5,8\}$ (there are many 3-element subsets that are mutually reachable from each other under the action of the semigroup, therefore they have isomorphic permutator groups). Computer calculation shows $|Perm(\{3,5,8\})| = 549$. Consider the following words of length 13 and 15, found by a breadth-first search, $a = t_1t_5t_3t_8t_5t_1t_4t_8t_5t_7t_8t_5t_6$, $b = t_1t_4t_8t_5t_3t_8t_5t_1t_4t_8t_5t_7t_8t_5t_6$ realizing transformations

$$
\begin{aligned}
\overrightarrow{a} &= ([1,2,10;\mathbf{3}],[4,7,9,11,12,15,16;\mathbf{5}],[6,13,14;\mathbf{8}])\\
\overrightarrow{b} &= [1,2,4;3]([10,11,12,13,14,15,16;\mathbf{5}],[6,7,9;\mathbf{8}]).
\end{aligned}
$$

As highlighted, these are clearly permutator words for the set $\{3,5,8\}$ and generate S_3. It is easy to verify that these two words are straight. Moreover, a and b can be checked to be minimal permutators (i.e. they cannot be properly factored

into permutators of $\{3, 5, 8\}$). However, the idempotent powers of these words $\overrightarrow{bb}$ and $\overrightarrow{aaa}$ are not equal, so the transformations do not lie in the same subgroup of the semigroup of the automaton.

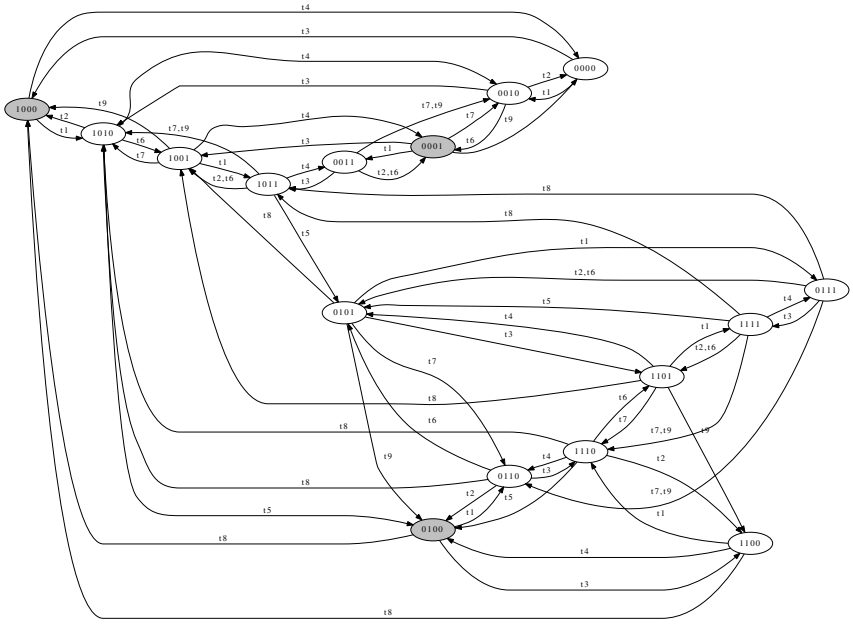


Fig. 2. Automaton derived from 2-level Petri net of the p53 system (16 states). The labels on the nodes encode the possible configurations for M, C, P and R (in this order). 0 denotes the absence (or presence below a threshold), 1 the presence (above the threshold) of the given type of molecule. For instance, 0101 means that C and R are present. The shaded states correspond to the state set $\{3 = 1000, 5 = 0001, 8 = 0100\}$.

We derive from these words, $x = bbabb$ (a word with 73 letters), which reduces to straight word: bba, (with only 43 letters), giving the transformation $\overrightarrow{x} = ([1, 2, 4; 5], [6, 7, 9; 8], [10, 11, 12, 13, 14, 15, 16; 3])$ and $y = aaabaaa$, (another long word with 93 letters), which reduces to straight word $aaab$ (with 54 letters), giving the transformation $\overrightarrow{y} = ([1, 2, 10; 5], [6, 13, 14; 8])[4, 7, 9, 11, 12, 15, 16; 3]$.

These words (a,b, $aaab$, bba) are all straight permutator words, but obviously $aaab$ and bba are not minimal permutators since they are products of (straight) minimal permutators a and b.

We have two copies of the symmetric group S_3 each faithfully acting on $\{3, 5, 8\} = \{M, R, C\}$: one S_3 is generated by a and y, and another isomorphic copy of S_3 by b and x with idempotents (the identity elements of these two groups): $\overrightarrow{a^3} = \overrightarrow{y^2} = [1, 2, 10; 8][4, 7, 9, 11, 12, 15, 16; 3][6, 13, 14; 5]$ and $\overrightarrow{b^2} = \overrightarrow{x^3} = [1, 2, 4; 3][6, 7, 9; 5][10, 11, 12, 13, 14, 15, 16; 8]$, respectively.

Together the elements $\overrightarrow{a}$ and $\overrightarrow{b}$ generate a 12 element semigroup which is just the union of these two groups. This is to some extent counterintuitive, as one

would expect one copy of the permutator group for one particular subset of the state set; furthermore as mentioned above the permutator semigroup $Perm(Y)$ has, not just these 12, but 549 elements, and this is but one of many instances of sets of states in this biological model acted on by the symmetric group S_3.

5 Conclusion

Based on algorithmic efficiency considerations we studied straight words that encode loop-free paths in the Cayley graph of a transformation semigroup. We focused on straight words generating transformations that permute a given subset of the state set. We found that these minimal permutator straight words form a finite code, and also the minimal permutator words form a code, although, as easy examples show, the latter is generally an infinite code. The minimal permutator straight words generate the corresponding subgroup of the transformation semigroup. These can be exploited in the calculations of hierarchical decompositions. These findings show that there may be a lot more ways within a semigroup to generate a subgroup than one might think, but for finding the subgroup it is enough to consider a subset of them.

References

1. Egri-Nagy, A., Nehaniv, C.L.: Algebraic hierarchical decomposition of finite state automata: Comparison of implementations for Krohn-Rhodes Theory. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 315–316. Springer, Heidelberg (2005)
2. Egri-Nagy, A., Nehaniv, C.L.: Cycle structure in automata and the holonomy decomposition. Acta Cybernetica 17, 199–211 (2005) ISSN: 0324-721X
3. Egri-Nagy, A., Nehaniv, C.L.: Algebraic properties of automata associated to Petri nets and applications to computation in biological systems. BioSystems 94(1-2), 135–144 (2008)
4. Egri-Nagy, A., Nehaniv, C.L.: `SgpDec` – software package for hierarchical coordinatization of groups and semigroups, implemented in the `GAP` computer algebra system, Version 0.5.24+ (2010), `http://sgpdec.sf.net`
5. Ganyushkin, O., Mazorchuk, V.: Classical Transformation Semigroups. Algebra and Applications. Springer, Heidelberg (2009)
6. The GAP Group: GAP – Groups, Algorithms, and Programming, Version 4.4 (2006), `http://www.gap-system.org`
7. Kastan, M.B., Kuerbitz, S.J.: Control of G1 arrest after DNA damage. Environ. Health Perspect. 101(Suppl. 5), 55–58 (1993)
8. Krohn, K., Rhodes, J.L., Tilson, B.R.: The prime decomposition theorem of the algebraic theory of machines. In: Arbib, M.A. (ed.) Algebraic Theory of Machines, Languages, and Semigroups, ch. 5, pp. 81–125. Academic Press, London (1968)
9. Rhodes, J.: Applications of Automata Theory and Algebra via the Mathematical Theory of Complexity to Biology, Physics, Psychology, Philosophy and Games. World Scientific Press, Singapore (2009)

On Lazy Representations and Sturmian Graphs*

Chiara Epifanio[1], Christiane Frougny[2], Alessandra Gabriele[1], Filippo Mignosi[3], and Jeffrey Shallit[4]

[1] Dipartimento di Matematica e Informatica, Università di Palermo, Italy
{epifanio,sandra}@math.unipa.it
[2] LIAFA, CNRS & Université Paris 7, and Université Paris 8, France
Christiane.Frougny@liafa.jussieu.fr
[3] Dipartimento di Informatica, Università di L'Aquila, Italy
mignosi@di.univaq.it
[4] School of Computer Science, University of Waterloo, Ontario, Canada
shallit@graceland.math.uwaterloo.ca

Abstract. In this paper we establish a strong relationship between the set of lazy representations and the set of paths in a Sturmian graph associated with a real number α. We prove that for any non-negative integer i the unique path weighted i in the Sturmian graph associated with α represents the lazy representation of i in the Ostrowski numeration system associated with α. Moreover, we provide several properties of the representations of the natural integers in this numeration system.

Keywords: numeration systems, Sturmian graphs, continued fractions.

1 Introduction

In [6] the authors have defined a new structure, the *Sturmian graph* associated with the continued fraction expansion of a real number α. They have also proved that Sturmian graphs have a counting property. In particular, given an infinite Sturmian graph, it can "count" from 0 up to infinity, which means that, for any $i \in \mathbb{N}$, there exists in this Sturmian graph a unique path starting in the initial state having weight i. Recent results on Sturmian graphs and Sturmian words and their generalizations can be found in [2]. The counting property proved in [6] has suggested us to introduce a continued fraction expansion-based numbering, a kind of numeration system that has strong relationships with Sturmian graphs. Despite this link, the new theory that we will introduce in this paper can be described in an independent way. Anyway, we will prove that it is well describable even through Sturmian graphs. We show that there exists a relation between the set of paths in a Sturmian graph and the set of lazy representations. In particular, we prove that for any number i the unique path weighted i in the Sturmian graph associated with α represents the lazy representation in the Ostrowski numeration system associated with α.

* Partially supported by MIUR National Project PRIN "Aspetti matematici e applicazioni emergenti degli automi e dei linguaggi formali".

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 125–134, 2011.

The paper is organized as follows. In the next section, we recall some basic notation on continued fraction expansions of a real number α and introduce Sturmian graphs. In the third section we focus on the representations of the natural integers in numeration systems defined by a basis. In the fourth section we focus on the relationship between the continued fraction expansion of a real number α and numeration systems. In the fifth section we explicit the link between a set of representations, called lazy, and Sturmian graphs. Finally, the last section contains some conclusions.

2 Continued Fraction Expansions, Sturmian Words and Sturmian Graphs

If α is a real number, we can expand α as a *simple continued fraction*

$$\alpha = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \dots}}},$$

which is usually abbreviated as $\alpha = [a_0, a_1, a_2, a_3, \dots]$, where a_0 is some integer ($a_0 \in \mathbb{Z}$) and all the other numbers a_i are positive integers. The integers in the continued fraction expansion of a real number are called *partial quotients*. The expansion may or may not terminate. If α is irrational, this representation is infinite and unique. If α is rational, there are two possible finite representations. Indeed, it is well known that $[a_0, \dots, a_{s-1}, a_s, 1] = [a_0, \dots, a_{s-1}, a_s + 1]$. For references to continued fractions, see [4], [8, Chap. 10], [12], and [13]. In this paper, we only discuss the case where a_i is a positive integer for $i \geq 0$, i.e., α is greater than or equal to 1. Moreover, we will focus on infinite continued fraction expansions. Hence, in our paper α will be an irrational number.

Given the continued fraction expansion of α, it is possible to construct a sequence of rationals $\frac{P_i(\alpha)}{Q_i(\alpha)}$, called "convergents", that converges to α, as well as a sequence of natural numbers $U_\alpha = (l_i)_{i\geq 0}$ such that $l_{i+1} = P_i + Q_i$, for any $i \geq 0$. They are defined by the following rules, with $P_i = P_i(\alpha)$ and $Q_i = Q_i(\alpha)$

$$\begin{array}{lll} P_0 = a_0 & Q_0 = 1 & l_0 = 1 \\ P_1 = a_1 a_0 + 1 & Q_1 = a_1 & l_1 = a_0 + 1 \\ P_{i+1} = a_{i+1} P_i + P_{i-1} & Q_{i+1} = a_{i+1} Q_i + Q_{i-1} & l_{i+1} = a_i l_i + l_{i-1} \quad i \geq 1 \end{array} \tag{1}$$

Notice that for $i \geq 0$, $P_i/Q_i = [a_0, \dots, a_i]$ and that under our assumptions a_i is a positive integer for $i \geq 0$ and hence the sequence $U_\alpha = (l_i)_{i\geq 0}$ is increasing.

As well-known, simple continued fractions play a leading role in the construction of Sturmian words. Indeed, among the different definitions of Sturmian words, one is obtained by applying the *standard method*. For references on Sturmian words and their geometric representation see [5], [9], [11, Chap. 2].

In [6] authors have defined a new structure, the *Sturmian graph* associated with the continued fraction expansion of a real number, and proved that these

graphs turn out to be the underlying graphs of compact directed acyclic word graphs of central Sturmian words (see [6] for further details). In this paper we want to deepen the structure of Sturmian graphs in order to establish a connection between them and a particular set of representations of non-negative integers. In order to be self-contained, we give in this paper a direct definition of the semi-normalized infinite Sturmian graph $G'(\alpha)$ associated with a real number α, that can be easily derived from definitions in [6].

Definition 1. *The* semi-normalized infinite Sturmian graph associated with a real number α, $G'(\alpha)$, *is a weighted semi-normalized infinite graph where each state has outgoing degree* 2. *Moreover if we number each state, the initial one having number* 0, *the arcs are defined in the following way.*

For any $s \geq 0$, *let* $b_s = \sum_{h=0}^{s} a_h$ *and, for any* $i \geq 0$, *let* $s(i)$ *be the smallest integer such that* $i < b_{s(i)}$. *Then the state numbered* i *has an outgoing arc weighted* $l_{s(i)}$ *to state numbered* $i+1$ *and an outgoing arc weighted* $l_{s(i)+1}$ *to state numbered* $1 + b_{s(i)}$.

Example 1. Let us consider the irrational number $\alpha = [1, 1, 2, 1, 2, 1, 2, \ldots] = \sqrt{3}$. Sequences $(l_i)_{i\geq 0}$, $(s(i))_{i\geq 0}$, $(b_{s(i)})_{i\geq 0}$ associated with this sequence are described in the following table.

i	0	1	2	3	4	5	6	7	$\cdots$
a_i	1	1	2	1	2	1	2	1	$\cdots$
b_i	1	2	4	5	7	8	10	11	$\cdots$
l_i	1	2	3	8	11	30	41	112	$\cdots$
$s(i)$	0	1	2	2	3	4	4	5	$\cdots$
$b_{s(i)}$	1	2	4	4	5	7	7	8	$\cdots$

Hence, the semi-normalized infinite Sturmian graph $G'(\alpha)$ is the following one

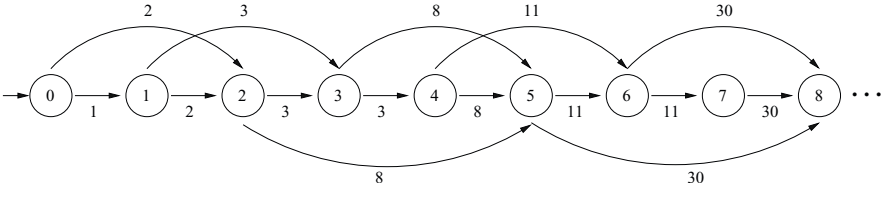


Fig. 1. The semi-normalized infinite Sturmian graph $G'([1, 1, 2, 1, 2, 1, 2, \ldots])$

Remark 1. Notice that the graphs of Definition 1 are exactly the semi-normalized infinite Sturmian graphs defined in [6]. Indeed, given a real number α, both definitions lead to the same graph $G'(\alpha)$ that is the limit graph of the sequence $G'(\frac{P_n}{Q_n})$, $n \in \mathbb{N}$, where $\frac{P_n}{Q_n}$ is the sequence of convergents of α and $G'(\frac{P_n}{Q_n})$ is defined in [6, Remark 6].

Let us analyze the structure of the semi-normalized infinite Sturmian graphs $G'(\alpha)$. We start with a proposition that characterizes the weights of arcs ingoing a given state.

Proposition 1. *For any $h \geq 1$, every arc ingoing the states in the set $S_h = \{b_{h-1}+1, b_{h-1}+2, \ldots, b_h\}$ has weight l_h and they are the unique arcs having this weight. If $h = 0$ then arcs ingoing the states in the set $S_0 = \{1, \ldots, b_h\}$ have weight l_h and they are the unique arcs having this weight.*

Remark 2. For any $h \geq 0$ the cardinality of the set S_h is obviously a_h.

Next propositions, as well as being interesting in themselves, are important as they will also be useful in the final section where we will show the strong relationship between a particular set of representations of non-negative integers, called *lazy representations*, and the Sturmian graphs.

Proposition 2. *Let us consider an arc (i, j). If there exists a number $h \geq 0$ such that i is smaller than $b_{h-1}+1$ and j is greater than b_h then $i = b_{h-1}$ and $j = b_h + 1$.*

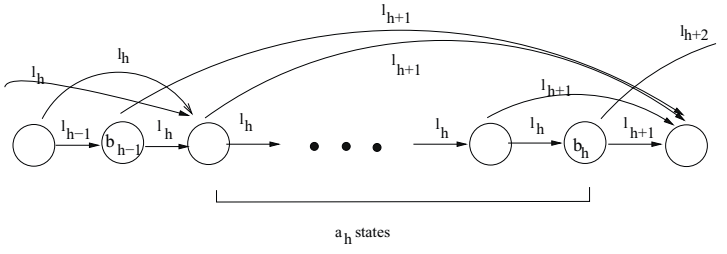


Fig. 2. The local structure of $G'(\alpha)$

Proposition 3. *For any $h \geq 0$ there exist no paths in the graph having more than a_h arcs weighted by l_h. Moreover, if a path from the initial state reaches state b_h then it contains exactly a_h arcs weighted by l_h.*

Definition 2. *Let us suppose we have an infinite graph G where the states are labeled by the integers greater than or equal to 0. Graph G can be weighted or non-weighted. We say that G is eventually periodic if there exist integer $p > 0$, called the* period*, and $\hat{n} \geq 0$ such that for any $n \geq \hat{n}$ one has that (n, j) is an arc of G if and only if $(n + p, j + p)$ is an arc of G.*

Remark 3. Def. 2 states a property that leaves weights out of consideration.

Proposition 4. *$G'(\alpha)$ is eventually periodic if and only if the continued fraction expansion of α is eventually periodic.*

3 Numeration Systems and Lazy Representations of Integers

This section is devoted to the representations of non-negative integers in numeration systems defined by an increasing basis. In particular, we focus on greedy

and lazy representations and we give a new algorithm for finding the lazy representation of a non-negative integer N.

The definitions we use are not the standard ones. Indeed our definitions are chosen among all possible "classical" ones because of our new point of view that let lazy representations be linked with Sturmian graphs.

Classically speaking, a numeration system is defined by a pair composed of either a *base* or a *basis*, which is an increasing sequence of numbers, and of an alphabet of digits. Standard numeration systems, such as the binary and the decimal ones, are represented in the first manner, i.e., through a base, while we are interested in the second one, i.e., through a basis. The reader may consult the survey [11, Chapt. 7]. More formally, we give the following definitions.

Definition 3. *Let $U = (u_i)_{i \geq 0}$ be a increasing sequence of integers with $u_0 = 1$, the* basis. *A U-representation of a non-negative integer N is a word $d_k \cdots d_0$ where the digits d_i, $0 \leq i \leq k$, are integers, such that $N = \sum_{i=0}^{k} d_i u_i$.*

Set $c_i = \lceil \frac{u_{i+1}}{u_i} \rceil - 1$. The U-representation $d_k \cdots d_0$ is said to be legal *if for any i, $0 \leq i \leq k$, one has that $0 \leq d_i \leq c_i$. The set $A = \{c \in \mathbb{N} \mid \exists i, 0 \leq c \leq c_i\}$ is the* canonical alphabet. *By convention, the representation of 0 is ϵ.*

Even if it is a very natural concept, the definition of legal U-representation is new. The concept of legal U-representation represents an essential requisite of our theory, because it allows us to link lazy representations with Sturmian graphs. In the following we will prove several properties of this representations.

Example 2. Let $F = (F_n)_{n \geq 0} = 1, 2, 3, 5, 8, 13, 21, 34, \ldots$ be the sequence of Fibonacci numbers obtained inductively in the following way: $F_0 = 1, F_1 = 2, F_{n+1} = F_n + F_{n-1}, n \geq 1$. The canonical alphabet is equal to $A = \{0, 1\}$. It is the well-known Fibonacci numeration system. An F-representation of the number 31 is 1010010. Another representation is 1001110. By definition, every F-representation of a non-negative integer N over $A = \{0, 1\}$ is legal.

Among all possible U-representations of a given non-negative integer N, one is known as the *greedy* (or *normal*) U-representation of N. It is the largest one in the lexicographic order.

Definition 4. *A* greedy *(or* normal*) U-representation of a given non-negative integer N is the word $d_k \cdots d_0$, where the most significant digit $d_k > 0$ and $d_j \geq 0$ for $0 \leq j < k$, and satisfying for each i, $0 \leq i \leq k$, $d_i u_i + \cdots + d_0 u_0 < u_{i+1}$.*

Now we consider another peculiar U-representation, linked to greedy representations, that is called the *lazy U-representation* of a natural number N.

Definition 5. *A word $e_k \cdots e_0$, with the most significant digit $e_k > 0$, is the* lazy U-representation *of a natural number N if it is the smallest legal U-representation of N in the radix order.*

Example 3. If we consider the Fibonacci numeration system of previous examples, the lazy F-representation of the number 31 is 111110.

Definition 6. *Let $w = d_k \cdots d_0$ be a U-representation. Denote $\underline{d_i} = c_i - d_i$, and by extension, $\underline{w} = \underline{d_k} \cdots \underline{d_0}$ the complement of w.*

By using the previous definition, we can link the greedy and lazy U-representations of a natural number N. For $k \geq 0$ set $C_k = \sum_{i=0}^{k} c_i u_i$.

Proposition 5. *A U-representation w of a number N, $u_k \leq N < u_{k+1}$, is greedy if and only if its complement $\underline{w}$ is the lazy U-representation of the number $N' = C_k - N$, up to eliminating all the initial zeros.*

Proposition 5 allows us to characterize the lazy U-representation of N.

Corollary 1. *A U-representation $e_k \cdots e_0$ of a number N is lazy if and only if for each i, $0 \leq i \leq k$, $e_i u_i + \cdots + e_0 u_0 > C_i - u_{i+1}$.*

Let us denote by m_i the greatest in the radix order of greedy U-representations of length i. Clearly m_i is the greedy representation of the integer $u_i - 1$. Recall that $m_0 = \varepsilon$. Denote by $M(U) = \{m_i \mid i \geq 0\}$.

Proposition 6. *1. A U-representation $w = d_k \cdots d_0$ of a natural number N is greedy if and only if for any i, $0 \leq i \leq k$, $d_i \cdots d_0 \leq m_{i+1}$ (in the radix order).*
2. A U-representation $w = d_k \cdots d_0$ of a natural number N is lazy if and only if for any i, $0 \leq i \leq k$, $\underline{d_i} \cdots \underline{d_0} \leq m_{i+1}$ (in the radix order).

A direct consequence of Proposition 6 is the following result.

Corollary 2. *For each $i \geq 0$ the number $u_i - 1$ has a unique legal U-representation.*

Now we are ready to give an algorithm computing lazy U-representations.

Lazy algorithm
Let $k = k(N)$ be the integer such that $C_{k-1} < N \leq C_k$. This ensures that the length of the lazy U-representation of N is $k + 1$.

Compute a U-representation $d_k \cdots d_0$ of $N' = C_k - N$ by the following algorithm: let $d_k = q(N', u_k)$ and $r_k = r(N', u_k)$, and, for $i = k - 1, \ldots, 0$, $d_i = q(r_{i+1}, u_i)$ and $r_i = r(r_{i+1}, u_i)$. Then $d_k \cdots d_0$ is a greedy U-representation of N' with possibly initial zeros.

By Proposition 5, the lazy U-representation of N is $\underline{d_k} \cdots \underline{d_0}$.

Denote by $\mathsf{Greedy}(U)$ and by $\mathsf{Lazy}(U)$ the sets of greedy and lazy U-representations of the non-negative integers. The regularity of the set $\mathsf{Greedy}(U)$ has been extensively studied. The following result is in [7, Prop. 2.3.51, Prop. 2.6.4].

Proposition 7. *The set $\mathsf{Greedy}(U)$ is regular if and only if the set $M(U)$ of greatest U-representations in the radix order is regular.*

Then by Proposition 6 Item 2 follows the following result.

Proposition 8. *The set $\mathsf{Lazy}(U)$ is regular if and only if the set $\mathsf{Greedy}(U)$ is regular if and only if the set $M(U)$ of greatest U-representations in the radix order is regular.*

4 Ostrowski Numeration System and Lazy Representations

In this section we are interested in the relationship between the continued fraction expansion of a real number α and numeration systems. Let us go into details by first recalling a numeration system, originally due to Ostrowski, which is based on continued fractions, see [1, p. 106] and [3]. This numeration system, called Ostrowski numeration system, can be viewed as a generalization of the Fibonacci numeration system.

Definition 7. *The sequence $(Q_i(\alpha))_{i\geq 0}$, defined in (1), of the denominators of the convergents of the infinite simple continued fraction of the irrational $\alpha = [a_0, a_1, a_2, \ldots] > 0$ forms the basis of the* Ostrowski numeration system based on α.

Proposition 9. *Let $\alpha = [a_0, a_1, a_2, \ldots] > 0$. The sequence $U_\alpha = (l_i)_{i\geq 0}$ associated in (1) with α is identical to the sequence $(Q_i)_{i\geq 0} = (Q_i(\beta))_{i\geq 0}$ defined in (1) for the number $\beta = [b_0, b_1, b_2, b_3, \ldots] = [0, a_0 + 1, a_1, a_2, \ldots]$.*

It is easy to verify that there exists a relation between β and α.

Proposition 10. *If α is greater than or equal to 1 then $\beta = \frac{1}{\alpha+1}$.*

In what follows, $\alpha = [a_0, a_1, \ldots]$ is greater than 1 and thus the sequence $U_\alpha = (l_i)_{i\geq 0}$ is increasing. In view of Definition 7 and Proposition 9, the Ostrowski numeration system based on $\beta = \frac{1}{\alpha+1}$ and the numeration system with basis U_α, in the sense of Definition 3, coincide. So we call the numeration system with basis U_α the *Ostrowski numeration system associated with* α.

By Definition 3, a U_α-representation $d_k \cdots d_0$ is legal if $d_i \leq a_i$, for any i such that $0 \leq i \leq k$, since in this case $c_i = a_i = \lceil \frac{l_{i+1}}{l_i} \rceil - 1$, thus the canonical alphabet is $A = \{a \in \mathbb{N} \mid \exists i, 0 \leq a \leq a_i\}$.

The Ostrowski numeration system associated with the golden ratio φ is the Fibonacci numeration system defined in Example 2. It is folklore that a U_φ-representation of an integer is greedy if and only if it does not contain any factor of the form 11, and is lazy if and only if it does not contain any factor of the form 00. We now extend this property to U_α-representations for any $\alpha > 1$. The greedy case is classical, see [1].

Proposition 11. *1. A U_α-representation $w = d_k \cdots d_0$ is greedy if and only if it contains no factor $d_i d_{i-1}$, $1 \leq i \leq k$, with $d_i = a_i$ and $d_{i-1} > 0$.*
2. A U_α-representation $w = d_k \cdots d_0$ is lazy if and only if it contains no factor $d_i d_{i-1}$, $1 \leq i \leq k$, with $d_i = 0$ and $d_{i-1} < a_{i-1}$.

Now we give a characterization of m_{i+1}, the greatest in the radix order of greedy U_α-representations of length $i + 1$.

Lemma 1. *For any $i \geq 0$, $m_{i+1} = a_i 0 a_{i-2} 0 \cdots a_2 0 a_0$ if i is even, and $m_{i+1} = a_i 0 a_{i-2} 0 \cdots a_1 0$ if i is odd.*

The sequence $(a_i)_{i\geq 0}$ is *eventually periodic* if there exist integers $m \geq 0$ and $p \geq 1$ such that $a_{i+p} = a_i$ for $i \geq m$. It is a classical result that the sequence $(a_i)_{i\geq 0}$ is eventually periodic if and only if α is a quadratic irrational.

The regularity of the set of the greedy U_α-representations has been already studied. In particular, it is proved in Shallit [14] and Loraud [10] that the set of greedy expansions in the Ostrowski numeration system associated with $\alpha > 1$ is regular if and only if the sequence $(a_i)_{i\geq 0}$ is eventually periodic.

Lemma 2. *The set $M(U_\alpha)$ is regular if and only if the sequence $(a_i)_{i\geq 0}$ is eventually periodic.*

Next Proposition follows from Proposition 8 and Lemma 2.

Proposition 12. *The sets of greedy expansions and of lazy expansions in the Ostrowski numeration system associated with $\alpha > 1$ are regular if and only if the sequence $(a_i)_{i\geq 0}$ is eventually periodic if and only if α is a quadratic irrational.*

5 Sturmian Graphs and Lazy Representations

The goal of this section is to establish a deep connection between the set of lazy representations and the set of paths in a well defined *Sturmian graph*.

We start by recalling a classical definition.

Definition 8. *Let G be a weighted graph, the weight of a path in G is the sum of the weights of all arcs in the path.*

Epifanio *et al.* [6] have proved several properties on finite and infinite Sturmian graphs. Among them, an important result regards a *counting property*. Concerning infinite Directed Acyclic Graphs (DAGs), this property can be stated in the following way.

Definition 9. *An infinite semi-normalized weighted DAG G' has the $(h, +\infty)$-*counting property*, or, in short, counts from h to $+\infty$, if any non-empty path starting in the initial state has weight in the range $h \cdots +\infty$ and for any i, $i \geq h$, there exists a unique path that starts in the initial state and has weight i.*

Starting from this definition, at the end of the proof of Proposition 35 in [6] it has been proved the following result concerning the counting property of infinite Sturmian graphs, that is very useful in the next.

Theorem 1. *For any positive irrational α, $G'(\alpha)$ can count from 0 up to infinity.*

Moreover, we can prove the following result that characterizes the path weights of states in $G'(\alpha)$.

Proposition 13. *For any state $i > 1$ in $G'(\alpha)$ let b_s be the maximum non-negative integer such that $i > b_s$ (cf. Def. 7). The maximum weight of the paths from the initial state ending in i is $g(i) = \sum_{j=0}^{s} a_j l_j + (i - b_s) l_{s+1}$. The only paths from the initial state ending in state $i = 0$ ($i = 1$ resp.) have weights 0 (1 resp.).*

Corollary 3. *For any state i in $G'(\alpha)$, all paths from the initial state ending in i are weighted N, where N is such that $g(i-1)+1 \leq N \leq g(i)$ and, conversely, if N is such that $g(i-1)+1 \leq N \leq g(i)$ then N is the weight of a path ending in i.*

Example 4. Let us consider the irrational number $\alpha = [1,1,2,1,2,1,2,\ldots]$ and its semi-normalized infinite Sturmian graph $G'(\alpha)$ represented in Figure 1. The following table represents, for any state i in $G'(\alpha)$, values b_s and s of Proposition 13, as well as the minimum, min_i, and the maximum, max_i, non-negative integers among the weights of all paths from the initial state ingoing in i.

i	0	1	2	3	4	5	6	7	$\cdots$
b_s			1	2	2	4	5	5	$\cdots$
s			0	1	1	2	3	3	$\cdots$
min_i	0	1	2	4	7	10	18	29	$\cdots$
max_i	0	1	3	6	9	17	28	39	$\cdots$

Before coming to the main result of the paper, we define the correspondence between paths and representations.

Definition 10. *Let $G'(\alpha)$ be the semi-normalized Sturmian graph associated to α and U_α be the Ostrowsky numeration system associated with α. For any $N \in \mathbb{N}$, we say that the U_α representation $d_k \cdots d_0$ of N* corresponds *to the unique path weighted N in $G'(\alpha)$ if, for any $i \geq 0$, d_i represents the number of consecutive arcs labeled Q_i in the path.*

Theorem 2. *Let N be a non-negative integer. A U_α-representation of N is lazy if and only if it corresponds to the unique path weighted N in the semi-normalized Sturmian graph $G'(\alpha)$.*

More precisely, the importance of this result lies on the fact that for any number i, we can connect the unique path weighted i in the Sturmian graph associated with α and the lazy representation of i in the Ostrowski numeration system associated with α.

Example 5. For $\alpha = [1,1,2,1,2,1,2,\ldots]$ the lazy U_α-representation of 7 is equal to 201. Recall that in this case $U_\alpha = \{1,2,3,8,11,30,41,112,\ldots\}$. Then 201 gives the decomposition $7 = 1 + 2 \cdot 3$.

On the other hand, given the graph $G'(\alpha)$ of Figure 1, there exists a unique path starting from 0 with weight 7. It is labelled 1, 3, 3, that exactly corresponds to the lazy U_α-representation of $7 = 1 + 2 \cdot 3$.

Next corollary is an immediate consequence of previous theorem, of Proposition 4 and of Proposition 12.

Corollary 4. *The set of all the U_α-representations that correspond to paths in $G'(\alpha)$ is regular if and only if $G'(\alpha)$ is eventually periodic.*

6 Conclusions

This paper contains a neat and natural theory on lazy representations in Ostrowski numeration system based on the continued fraction expansion of a real number α. This theory provides a natural understanding of the Sturmian graph associated with α, even better the study of Sturmian graphs gave us the idea to formalize and to develop this theory. Indeed, the set of lazy representations is naturally linked with the set of paths in the Sturmian graph associated with α. It would be interesting to deepen this theory in order to prove other properties of the representations in the Ostrowski numeration system based on the continued fraction expansion of a real number α through the Sturmian graph associated with α. Moreover, it would be nice to find algorithms for performing the elementary arithmetic operations.

References

1. Allouche, J.-P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press, Cambridge (2003)
2. Berstel, J.: Sturmian and episturmian words (a survey of some recent results). In: Bozapalidis, S., Rahonis, G. (eds.) CAI 2007. LNCS, vol. 4728, pp. 23–47. Springer, Heidelberg (2007)
3. Berthé, V.: Autour du système de numération d'Ostrowski. Bull. Belg. Math. Soc. Simon Stevin 8, 209–239 (2001)
4. Brezinski, C.: History of continued fractions and Padé approximants. Springer Series in Computational Mathematics, vol. 12. Springer, Heidelberg (1991)
5. de Luca, A., Mignosi, F.: Some combinatorial properties of Sturmian words. TCS 136, 361–385 (1994)
6. Epifanio, C., Mignosi, F., Shallit, J., Venturini, I.: On Sturmian graphs. DAM 155(8), 1014–1030 (2007)
7. Frougny, C., Sakarovitch, J.: Number representation and finite automata. In: Berthé, V., Rigo, M. (eds.) Combinatorics, Automata and Number Theory Encycl. of Math. and its Appl., ch. 2, vol. 135, Cambridge University Press, Cambridge (2010)
8. Hardy, G.H., Wright, E.M.: An Introduction to the Theory of Numbers, 5th edn. Oxford University Press, Oxford (1989)
9. Klette, R., Rosenfeld, A.: Digital straightness – a review. DAM 139(1-3), 197–230 (2004)
10. Loraud, N.: β-shift, systèmes de numération et automates. Journal de Théorie des Nombres de Bordeaux 7, 473–498 (1995)
11. Lothaire, M.: Algebraic Combinatorics on Words. In: Encycl. of Math. and its Appl., vol. 90, Cambridge University Press, Cambridge (2002)
12. Perron, O.: Die Lehre von den Kettenbrüchen. B. G. Teubner, Stuttgart (1954)
13. Shallit, J.: Real numbers with bounded partial quotients. Ens. Math. 38, 151–187 (1992)
14. Shallit, J.: Numeration systems, linear recurrences, and regular sets. Inform. and Comput. 113, 331–347 (1994)

Symbolic Dynamics, Flower Automata and Infinite Traces

Wit Foryś[1], Piotr Oprocha[2,3], and Slawomir Bakalarski[4]

[1] Jagiellonian University, Institute of Computer Science, Łojasiewicza 6, 30-348 Kraków, Poland
forysw@ii.uj.edu.pl
[2] Departamento de Matemáticas Universidad de Murcia Campus de Espinardo, 30100 Murcia, Spain
[3] Faculty of Applied Mathematics, AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Kraków, Poland
oprocha@agh.edu.pl
[4] Jagiellonian University, Institute of Computer Science, Łojasiewicza 6, 30-348 Kraków, Poland
slawomir.bakalarski@ii.uj.edu.pl

Abstract. Considering a finite alphabet as a set of allowed instructions, we can identify finite words with basic actions or programs. Hence infinite paths on a flower automaton can represent order in which these programs are executed and a flower shift related with it represents list of instructions to be executed at some mid-point of the computation.

Each such list could be converted into an infinite real trace when an additional information is given, namely which instructions can be executed simultaneously (so that way we obtain a schedule for a process of parallel computation). In this paper we investigate when obtained in such a way objects (sets of infinite real traces) are well defined from the dynamical point of view and to what extent they share properties of underlying flower shifts.

1 Introduction

One of the methods of representation of parallel computing it to convert a word into a trace. This approach originated from the fundamental papers by Cartier and Foata [2] and Mazurkiewicz [11]. Later, these concepts attracted a lot of attention of researchers investigating traces from different points of view like e.g. combinatorics [3], parallel computing [8] or topology [10].

In [5] we introduced a framework, joining the main ideas of symbolic dynamics and theory of traces. We proved there some basic facts connecting the dynamics of a given shift with trace shift formed as its counter part representing a parallel computation. In [6] we investigated properties of sets of traces generated by minimal shifts (shifts having no proper subshifts). In [7] we extended the framework to obtain a similar model to bi-infinite sequences in symbolic dynamics, that is we defined and investigated shifts on bi-infinite traces.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 135–142, 2011.

Many problems appear to be quite complex and remain open. Even in the case of small alphabet it is rather hard to provide a full characterization of the dynamical properties of induced t-shift.

In the present paper we focus our attention on flower automata and flower shifts. The motivation is two-fold. First of all, flower automaton provides a quite nice interpretation of recognized words. Namely, each word can be viewed as a separate program and so each (finite) word accepted by flower automaton provides an order of execution of these programs (with repetitions of course). This description can be easily extended onto infinite paths on the automaton. Second motivation is that flower shifts are probably the simplest cases of sofic shifts. Then proper understanding on the situation within the class of flower shift can be the first step towards understanding of the structure of sets of traces arising from sofic systems.

2 Definitions and Notations

We assume that all basic concepts of combinatorics on words, automata and trace theory and symbolic dynamics are known. A more extensive treatment may be found in [1,4,9].

2.1 Words and Traces

Let Σ^* denote the set of all finite words over alphabet Σ. Σ^* with the concatenation of words forms a free monoid (the empty word is denoted 1). The set of nonempty words is denoted by Σ^+. Σ^ω denotes the set of all infinite words (sequences) over Σ.

Let us denote $I \subset \Sigma \times \Sigma$ a symmetric and irreflexive relation defined on Σ and call it *independence relation*. Its complement is denoted by D and called *dependence relation*. The relation I is extended to a congruence $\sim_I$ on Σ^* by putting $u \sim_I v$ if it is possible to transform u to v (equivalently v to u) by a finite number of swaps $ab \to ba$ of independent letters. A *trace* is any element of the quotient space $\mathbb{M}(\Sigma, I) = \Sigma^*/\sim_I$. If $t \in \Sigma^*$ or $t \in \mathbb{M}(\Sigma, I)$ then by $\mid t \mid_a$ and $alph(t)$ we denote the number of occurrences of the symbols $a \in \Sigma$ in t and the set of all symbols which occur in t, respectively.

A word $w \in \Sigma^*$ is in *Foata normal form*, if it is the empty word or if there exist an integer $n > 0$ and nonempty words $v_1, ..., v_n \in \Sigma^+$ (called *Foata steps*) such that $w = v_1.....v_n$ and any word v_i is composed of pairwise independent letters and v_i is minimal with respect to the lexicographic ordering. Additionally for any $i = 1, ..., n-1$ and for any letter $a \in \text{alph}(v_{i+1})$ there must exist a letter $b \in \text{alph}(v_i)$ such that $(a, b) \in D$. The above mentioned lexicographical ordering on Σ^* refers to an ordering on Σ which must be fixed. It can be proved that for any $x \in \Sigma^*$ there exists the unique $w \in [x]_{\sim_I}$ in the Foata normal form.

2.2 Subshifts

To introduce symbolic dynamical systems (called shift or subshifts) we endow Σ^ω with the following metric d. If $x = y$ then $d(x, y) = 0$ and otherwise $d(x, y) = 2^{-j}$

when j is the number of symbols in the longest common prefix of x and y. The metric space (Σ, d) is compact which means that every sequence has a convergent subsequence. Now, define a shift map $\sigma : \Sigma^\omega \to \Sigma^\omega$ by $(\sigma(x))_i = x_{i+1}$ where $(\cdot)_i$ denotes the i-th letter of a sequence. It is easy to observe that σ is continuous.

Any closed and σ-invariant (i.e. $\sigma(X) \subset X$) set $X \subset \Sigma$ is called *shift* or *subshift*. By the *full shift* over Σ we mean Σ^ω together with the map σ.

2.3 Infinite Traces and T-Shifts

For $w = (w_i)_{i\in\mathbb{N}} \in \Sigma^\omega$ the *dependence graph* $\varphi_{\mathbb{G}}(w) = [V, E, \lambda]$ is defined as follows. We put $V = \mathbb{N}$ and $\lambda(i) = w_i$ for any $i \in \mathbb{N}$. There exists an arrow $(i, j) \in E$, if and only if $i < j$ and $(w_i, w_j) \in D$. Let us denote the set of all possible dependence graphs (up to isomorphism of graphs) by $\mathbb{R}^\omega(\Sigma, I)$ and let $\varphi_{\mathbb{G}} : \Sigma^\omega \to \mathbb{R}^\omega(\Sigma, I)$ be a natural projection. We call elements of $\mathbb{R}^\omega(\Sigma, I)$ *infinite (real) traces*. Each dependence graph is acyclic and it induces well-founded ordering on $\mathbb{N}$. Then for any $v \in V$ the function $h : V \to \mathbb{N}$ is well defined:

$$\mathcal{P}(v) = \{n \in \mathbb{N} : \exists v_1, ..., v_n \in V, v_n = v, (v_i, v_{i+1}) \in E \text{ for } i = 1, ..., n-1\}$$
$$h(v) = \max \mathcal{P}(v).$$

By $F_n(t)$ we denote a word $w \in \Sigma^*$ consisting of all the letters from the n-th level of infinite trace $t \in \mathbb{R}^\omega(\Sigma, I)$, that is from the set $\{\lambda(v) \ : \ v \in V, h(v) = n\}$. It follows from the definition of dependence graph that for any infinite trace $t \in \mathbb{R}^\omega(\Sigma, I)$ the word $w = F_1(t) \dots F_n(t)$ is in Foata normal form with Foata steps given by $F_i(t)$ and $t = \varphi_{\mathbb{G}}(F_1(t)F_2(t)\dots)$. Then in the same way as it was done for Σ^ω we may endow $\mathbb{R}^\omega(\Sigma, I)$ with a metric $d_{\mathbb{R}}(s, t)$ where $d_{\mathbb{R}}(s, t) = 0$ if $s = t$ and if $s \neq t$ then $d_{\mathbb{R}}(s, t) = 2^{-j+1}$ where j is the maximal integer such that $F_i(t) = F_i(s)$ for $1 \leq i \leq j$. It is known that the metric space $(\mathbb{R}^\omega(\Sigma, I), d_{\mathbb{R}})$ is compact [10].

By a *full t-shift* we mean the metric space $(\mathbb{R}^\omega(\Sigma, I), d_{\mathbb{R}})$ together with continuous map $\Phi : \mathbb{R}^\omega(\Sigma, I) \to \mathbb{R}^\omega(\Sigma, I)$ defined for any $t \in \mathbb{R}^\omega(\Sigma, I)$, by the formula

$$\Phi(t) = \varphi_{\mathbb{G}}(F_2(t)F_3(t)\dots)$$

Similarly, as for shifts, a *t-shift* is referred to any compact and Φ-invariant subset of $\mathbb{R}^\omega(\Sigma, I)$. It was proved in [5] that from the dynamical systems point of view $(\mathbb{R}^\omega(\Sigma, I), \Phi)$ is equivalent to a shift of finite type (which means that dynamics of $(\mathbb{R}^\omega(\Sigma, I), \Phi)$ and (Σ^ω, σ) is to some extent similar). However, it usually happens that the $\varphi_{\mathbb{G}}$ image of a shift is not a t-shift and there are also t-shifts which cannot be obtained as images of shifts [5].

3 Flower Shifts

The simplest step from minimal (having no proper subshifts [6]) to non-minimal shifts is via flower shifts (they have dense sets of periodic orbits and a very clear mechanism behind the construction of all other elements of the shift).

The notion of a flower shift is closely connected with a flower automaton which is a universal construction of automaton recognizing a submonoid W^* of Σ^* where $W = \{w_1, w_2, \ldots, w_n\} \subset \Sigma^+$. The main feature of a flower automaton is that the set of its edges is the disjoint union of cycles, all starting and ending in a common state and labelled by words from W (for a more formal definition, e.g. see [1]).

Basing on the notion of a flower automaton the notion of a *flower shift* is defined as follows. Let $W = \{w_1, w_2, \ldots, w_n\} \subset \Sigma^+$ and $M = \max_{j=1,\ldots,n} |w_j|$. Define

$$Y = \bigcup_{k=0}^{M} \sigma^k(\{w_1 w_2 \ldots : w_i \in W \text{ for all } i\}).$$

The set Y together with shift function σ is said to be the *flower shift* generated by words $w_1, \ldots, w_n$.

Note that flower shifts are also known under the name *renewal systems* [13] or *finitely generated systems* [12].

Generally, a transformation of a shift X to the set of traces via $\varphi_{\mathbb{G}}$ results in a set of traces $\varphi_{\mathbb{G}}(X)$ which may be not closed nor invariant. As we will see, even if we restrict our attention to the class of flower shifts there is no any simple solution for the above problem. Consider the following example. In this example and latter on w^∞ denotes infinite iteration of the word w, that is a infinite word (sequence) $ww \ldots w \ldots$.

Example 1. *Let $\Sigma = \{a, b, c, d\}$ and $I = \{(a, b), (b, a), (a, c), (c, a)\}$. We put $w_1 = bad$, $w_2 = cb$, $w_3 = bcd$ and $w_4 = acd$ and denote by X the flower shift generated by w_1, w_2, w_3, w_4. Observe that*

$$\varphi_{\mathbb{G}}((cb)^n(bad)(cb)^\infty) = \varphi_{\mathbb{G}}(a(cb)^n bd(cb)^\infty) \longrightarrow \varphi_{\mathbb{G}}(a(cb)^\infty)$$

which implies that $\varphi_{\mathbb{G}}(X)$ is not closed.

We may generalize the above example to the following

Theorem 2. *Let X be a flower shift generated by words $w_1, \ldots, w_n$. Denote by $r(w_j)$ the rightmost letter of w_j. If there exists $a \in \Sigma$ such that:*

1. *$w_1 \sim au$ for some $u \in \Sigma^*$*
2. *$a \notin \operatorname{alph}(w_2) \subset I(a)$*
3. *for all $j \neq 2$ either $\operatorname{alph}(w_j) \subset \operatorname{alph}(w_2)$ or $r(w_j) \notin \operatorname{alph}(w_2)$, $r(w_j) \times \operatorname{alph}(w_2) \cap D \neq \emptyset$ for some letter $r(w_j) \in \operatorname{alph}(w_j)$*

then $\varphi_{\mathbb{G}}(X)$ is not closed.

Proof. Let $x_n = w_2^n w_1 w_2^\infty$, $t_n = \varphi_{\mathbb{G}}(x_n)$ and $q = \lim_{n\to\infty} t_n$ (it is easy to see that this limit exists). We claim that $q \notin \varphi_{\mathbb{G}}(X)$. Suppose that there exists $x \in X$ such that $\varphi_{\mathbb{G}}(x) = q$. There exists a sequence $\{j_i\}_{i=1}^\infty$ such that $x = z w_{j_1} w_{j_2} \ldots$ where z is a suffix of some word w_j.

Consider w_j such that $\mathrm{alph}(w_j) \setminus \mathrm{alph}(w_2) \neq \emptyset$. From 3. $r(w_j) \notin \mathrm{alph}(w_2)$ and $r(w_j) \times \mathrm{alph}(w_2) \cap D \neq \emptyset$.

First, observe that $r(w_j)$ does not occur in q for any j such that $\mathrm{alph}(w_j) \setminus \mathrm{alph}(w_2) \neq \emptyset$. It implies that the letter $r(w_j)$ is pushed to the righthand side if $n \to \infty$. Hence $\mathrm{alph}(z) \subset \mathrm{alph}(w_2)$ and a does not occur in z.

Now, let k be the smallest integer such that $j_k \neq 2$. If $\mathrm{alph}(w_{j_k}) \setminus \mathrm{alph}(w_2) \neq \emptyset$, then, exactly as in the above, a does not occur in w_{j_k} and consequently $\mathrm{alph}(q) \neq \mathrm{alph}(x)$ which contradicts the assumptions. In the opposite case, that is if $\mathrm{alph}(w_{j_k}) \subset \mathrm{alph}(w_2)$ then a does not occur in the prefix of x equal to $zw_2...w_2w_{j_k}$. Repeating this reasoning we exclude the latter a from the sequence x - a contradiction.

Let (Σ, D) be a dependence alphabet and $X \subset \Sigma^\omega$ be a shift. If for any positive integer i there exists j such that the following implication holds for any $x, y \in X$

$$x_{[0,j]} = y_{[0,j]} \Rightarrow F_i(\varphi_G(x)) = F_i(\varphi_G(y))$$

then X is said to have a bounded Foata step horizon.

In [5] we prove that the condition of bounded Foata step horizon is sufficient for $\varphi_{\mathbb{G}}(X)$ to be closed. However even for flower shifts it is not necessary. Let us consider the following example.

Example 3. *Let $\Sigma = \{a, b, c\}$ and $I = \{(a, b), (b, a), (a, c), (c, a)\}$. We put $w_1 = a$, $w_2 = cb$ and denote by X the flower shift generated by w_1, w_2. Observe that $\varphi_{\mathbb{G}}(X)$ consists of traces: $\varphi_{\mathbb{G}}(a^\infty)$, $\varphi_{\mathbb{G}}((cb)^\infty)$, $\varphi_{\mathbb{G}}((bc)^\infty)$ and $\varphi_{\mathbb{G}}(a^k(cb)^\infty)$ for any $k \in \mathbb{N}$. Hence $\varphi_{\mathbb{G}}(X)$ is closed. However sequences $(cb)^n a(cb)^\infty$ and $(cb)^n a^\infty$ forms a counterexample for bounded Foata horizon property for X.*

Theorem 4. *Let X be a flower shift and let $w_1, \ldots, w_n$ be words generating X. If there exist letters $a, b \in \mathrm{alph}(X)$ such that for any $j = 1, ..., n$ the following conditions hold:*

1. *$b \notin \mathrm{alph}(w_j) \subset I(a)$ or*
2. *there exist $u, v \in \Sigma^*$ such that $w_j = uav$, $\mathrm{alph}(u) \subset I(a)$, $\mathrm{alph}(v) \setminus I(a) \neq \emptyset$, $|u|_a = 0$, $|v|_b = 0$ and additionally $|u|_b > |v|_a + 1$ for some $j = 1..., n$*

then $\varphi_{\mathbb{G}}(X)$ is not invariant.

Proof. For any $t \in \varphi_{\mathbb{G}}(X)$ the following implication follows from the assumptions about words w_j.

$$b \in \mathrm{alph}(F_1(t)) \quad \Longrightarrow a \in \mathrm{alph}(F_1(t)).$$

Let us consider some $w_j = uav$ with $|u|_b > |v|_a + 1$. Denote $s = \varphi_{\mathbb{G}}(w_j^\infty)$. There exists an integer m such that
$a \notin F_m(s)$, $b \in F_m(s)$. Finally observe that $F_1(\Phi^{m-1}(s)) = F_m(s)$ and so $\Phi^{m-1}(s) \notin \varphi_{\mathbb{G}}(X)$.

The following theorem is presented without a proof.

Theorem 5. *Let X be a flower shift generated by $w_1, w_2 \in \Sigma^+$. If $\mathrm{alph}(w_1) \times \mathrm{alph}(w_2) \subset I$, then $\varphi_{\mathbb{G}}(X)$ is closed.*

Remark 6. *Suppose that $w = us$ and $F_1(w)\dots F_k(w)$ is a Foata normal form of w. If $F_1(s)\dots F_{k-1}(s)$ is a Foata normal form of s such that $F_i(s) = F_{i+1}(w)$, then u is a permutation of $F_1(w)$.*

Proof. It is enough to note that $us \sim_I uF_1(s)\dots F_{k-1}(s) \sim_I uF_2(w)\dots F_k(w)$. Namely $\mathrm{alph}(u) = \mathrm{alph}(F_1(w))$ and so $u \sim_I F_1(w)$.

In the following lemma we use, in a very restrictive form, catenation of a finite trace with an infinite one. We assume, that a finite trace s has the Foata normal form such that the last Foata step is a one letter and that this letter is dependent on all letters from the alphabet. Taking any infinite trace t the concatenation st could be characterized and defined for our purpose, as represented by an infinite sequence of Foata steps in which steps of s are followed by steps of t.

Lemma 7. *Let us assume that a word $w \in \Sigma^+$ is such that $(r(w), a) \in D$ for all $a \in \Sigma$. Then for all suffixes s of w the mapping $\gamma_s : \mathbb{R}^\omega(\Sigma, I) \to \mathbb{R}^\omega(\Sigma, I)$, $\gamma_s(t) := \varphi_{\mathbb{G}}(s)t$ is continuous.*

Proof. For simplicity identify s with the trace $\varphi_{\mathbb{G}}(s)$. Let $F_1(s)F_2(s)\dots F_m(s)$ be the Foata normal form of s. Then $F_m(s) = r(w)$ and so

$$F_k(st) = \begin{cases} F_k(s) & k \le m \\ F_{m-k}(t) & k > m \end{cases}$$

It implies continuity of γ_s.

Theorem 8. *Let X be a flower shift generated by words $w_1, \dots, w_m \in \Sigma^+$ such that $(r(w_i), x) \in D$ for $i = 1, \dots, m$ and for all $x \in \Sigma$. Put $U = \{u_1u_2\dots \mid u_i \in \{w_1, \dots, w_m\}\}$. Then $\varphi_{\mathbb{G}}(X)$ is closed provided that $\varphi_{\mathbb{G}}(U)$ is closed.*

Proof. Let $t_n = \varphi_{\mathbb{G}}(x_n)$ be a sequence in $\varphi_{\mathbb{G}}(X)$ convergent to some t. Every x_n is of the form $x_n = s_nu_n, u_n \in U$ and s_n is a suffix of some w_i. Since the number of suffixes is finite (going to a subsequence if necessary) we may assume that $x_n = su_n$ and thus t_n can be considered as a sequence in $\gamma_s(\varphi_{\mathbb{G}}(U))$. Since $\varphi_{\mathbb{G}}(U)$ is compact, then by Lemma 7, also $\gamma_s(\varphi_{\mathbb{G}}(U))$ is compact. It immediately implies that $t \in \gamma_s(\varphi_{\mathbb{G}}(U)) \subset \varphi_{\mathbb{G}}(X)$ and so the proof is completed.

Now we show that the case of $\Sigma = \{a, b\}$ is completely different than the case of alphabets with more than two elements. Hence assume $\Sigma = \{a, b\}$. If I is empty then $\varphi_{\mathbb{G}}(X) = X$ for any flower shift X over Σ (in fact this obvious observation is generally true, for any shift X).

If $(a, b) \in I$ then $\varphi_{\mathbb{G}}(X)$ consists of a single fixed point, provided X is a flower shift generated by the unique word $w \in \{a, b\}^+$. A flower shift generated by a one word over any finite alphabet is of course a minimal shift; namely it is a single orbit. When there are more words available, then the situation changes a little, however still a t-shift is generated.

Theorem 9. *If X is a flower shift generated by words $w_1, w_2, \ldots, w_n \subset \{a,b\}^+$, $n > 0$ then $\varphi_{\mathbb{G}}(X)$ contains at most three fixed points and at most countable family of points eventually fixed (i.e. $\lim_{n\to\infty} \Phi(t_n)$ exists and is a fixed point). In particular $\varphi_{\mathbb{G}}(X)$ is closed.*

Proof. The case of $n = 1$ is obvious. The case w_1, w_2 is covered by Theorem 5. For $n > 2$ the proof follows the same lines. Namely, for a sequence $x = sw_{i_1}w_{i_2}\ldots$ we have the following three possibilities:

1. $x \sim_I (ab)^\infty$
2. $x \sim_I (ab)^k b^\infty$ for some $k \geq 0$,
3. $x \sim_I (ab)^k a^\infty$ for some $k \geq 0$,

and so the proof easily follows.

Theorem 10. *Let $\Sigma = \{a,b\}$, $(a,b) \in I$ and let $w_1, \ldots, w_n \in \Sigma^+$. If X is a flower shift generated by words $w_1, \ldots, w_n \in \Sigma^+$, then $\varphi_{\mathbb{G}}(X)$ is invariant.*

Proof. For $t \in \varphi_{\mathbb{G}}(X)$ there exists $x \in X$ such that $t = \varphi_{\mathbb{G}}(x)$. We have to consider the following three cases.

1. if $F_1(t) = a$, then $w = a^\infty$ and $\Phi(t) = t \in \varphi_{\mathbb{G}}(X)$,
2. if $F_1(t) = b$ then $w = b^\infty$ and $\Phi(t) = t \in \varphi_{\mathbb{G}}(X)$,
3. if $F_1(t) = ab$ then, denoting $M = \max\{i|\ F_i(t) = ab\}$ let us consider two possibilities.

If M is infinite (among words w_i there exist w_1, w_2 such that $w_1 = a^k, w_2 = b^l$ or $w_1 \sim_I a^k b^l$) then the equality $\Phi(t) = t$ is obvious.

If $M < \infty$ then exactly M letters a occur in x. There exists the minimal number n such that $x_{[0,n]}$ contains exactly M letters a (it means that $x_{[n+1,\infty)} = b^\infty$.) Let m be a positive integer such that $x_{[0,m]}$ contains exactly one a. Of course $x_{[m+1,\infty)} \in X$ and

$$\begin{aligned}\Phi(t) &= \Phi(\varphi_{\mathbb{G}}(x)) = \Phi(\varphi_{\mathbb{G}}(x_{[0,m]}x_{[m+1,n]}b^\infty)) \\ &= \varphi_{\mathbb{G}}(x_{[m+1,n]}b^\infty) \in \varphi_{\mathbb{G}}(X).\end{aligned}$$

Example 11. *Suppose that $\Sigma = \{a,b,c\}$ and $(a,b) \notin I$, $(a,c) \in I$. Let $w_1 = a$, $w_2 = cb$. Denote by X a flower shift generated by these words. Let us consider $x_k = a^k cba^\infty \in X$. Then*

$$\lim_{k\to\infty} \varphi_{\mathbb{G}}(x_k) = \lim_{k\to\infty} \varphi_{\mathbb{G}}((ac)a^{k-1}ba^\infty) = \varphi_{\mathbb{G}}((ac)a^\infty) \notin \varphi_{\mathbb{G}}(X)$$

and so $\varphi_{\mathbb{G}}(X)$ is not closed.

Example 12. *Suppose that $\Sigma = \{a,b,c\}$ and I contains all possible pairs of letters. Put $w_1 = cbbaa$ and $w_2 = a$. Notice that $\varphi_{\mathbb{G}}(X)$ is not invariant, where X is a flower shift generated by w_1, w_2. For this consider $x = w_1 w_2^\infty$. $\varphi_{\mathbb{G}}(x)$ has the following Foata normal form $abcaba^\infty$ and $\varphi_{\mathbb{G}}(abcaba^\infty)$ is not in $\varphi_{\mathbb{G}}(X)$.*

Acknowledgements

This research was supported by the Polish Ministry of Science and Higher Education (2010). The research of P. Oprocha leading to results included in the paper has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 219212, belonging to a Marie Curie Intra-European Fellowship for Career Development. He was also supported and MICINN (Ministerio de Ciencia e Innovacion) and FEDER (Fondo Europeo de Desarrollo Regional), grant MTM2008-03679/MTM.

The financial support of these institutions is hereby gratefully acknowledged.

References

1. Berstel, J., Perrin, D.: Theory of codes, Pure and Applied Mathematics, vol. 117. Academic Press Inc., Orlando (1985)
2. Cartier, P., Foata, D.: Problèmes combinatories de commutation et réarrangements. Lecture Notes in Mathematics. Springer, Heidelberg (1969)
3. Diekert, V. (ed.): Combinatorics on Traces. LNCS, vol. 454. Springer, Heidelberg (1990)
4. Diekert, V., Rozenberg, G. (eds.): The book of traces. World Scientific Publishing Co. Inc., River Edge (1995)
5. Foryś, W., Oprocha, P.: Infinite Traces and Symbolic Dynamics. Theory Comput. Syst. 45, 133–149 (2009)
6. Foryś, W., Oprocha, P.: Infinite Traces and Symbolic Dynamics - Minimal Shift Case. Fundamenta Informatica (to appear)
7. Garcia Guirao, J.L., Foryś, W., Oprocha, P.: A dynamical model of parallel computation on bi-infinite time-scale. J.Comput. Appl. Math. (to appear)
8. Kaldewaij, A.: Trace theory and the specification of concurrent systems. In: Denvir, B.T., Jackson, M.I., Harwood, W.T., Wray, M.J. (eds.) ACS 1985. LNCS, vol. 207, pp. 211–221. Springer, Heidelberg (1985)
9. Kůrka, P.: Topological and Symbolic Dynamics. Cours Spécialisés [Specialized Courses] 11. Société Mathématique de France, Paris (2003)
10. Kwiatkowska, M.: A metric for traces. Inform. Process. Lett. 35, 129–135 (1990)
11. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. DAIMI Rep. Aarhus University 78, 1–45 (1977)
12. Restivo, A.: Finitely generated sofic systems. Theoret. Comput. Sci. 65, 265–270 (1989)
13. Williams, S.: Notes on renewal systems. Proc. Amer. Math. Soc. 110, 851–853 (1990)

The Cayley-Hamilton Theorem for Noncommutative Semirings

Radu Grosu

Department of Computer Science, Stony Brook University
Stony Brook, NY 11794-4400, USA

Abstract. The Cayley-Hamilton theorem (CHT) is a classic result in linear algebra over fields which states that a matrix satisfies its own characteristic polynomial. CHT has been extended from fields to commutative semirings by Rutherford in 1964. However, to the best of our knowledge, no result is known for noncommutative semirings. This is a serious limitation, as the class of regular languages, with finite automata as their recognizers, is a noncommutative idempotent semiring. In this paper we extend the CHT to noncommutative semirings. We also provide a simpler version of CHT for noncommutative idempotent semirings.

1 Introduction

The continuous dynamics of each mode of a linear hybrid automaton (HA) [2,7] is a linear system, as is the discrete switching logic among modes [5]. However, the former operates on a vector space, whereas the latter operates on a semimodule [4]. As a consequence, understanding which properties of linear systems hold in both vector spaces and semimodules, and which do not, is essential for developing a formal foundation and associated analysis tools for HAs.

Vector spaces (VSs) have a long history and consequently a large variety of analysis techniques. A defining aspect of a VS is the associated field of scalars, a structure possessing two operations, addition and multiplication, together with their identities and their inverses. Both are associative and commutative and multiplication distributes over addition. For example, $\mathbb{R}$ and $\mathbb{C}$ are fields.

Finding the fixpoint of the equation $x = Ax+B$ in a VS seems to be routine: $x = (I - A)^{-1}B$. However, this is far from being true. For large systems, computing the inverse of a matrix is expensive, and one often uses iterative Gauss-Seidel or Jacobi techniques [8]. Both are based on the identity $(I - A)^{-1} = A^*$, where $A^* = \sum_{n\in\mathbb{N}} A^n$, and converge if the eigenvalues of A are in the unit disc of $\mathbb{C}$.

The above solution for x leads to an amazing conclusion: fixpoint computation does not require inverses or commutativity of multiplication. Hence, one may consider a weaker structure, that lacks subtraction and division, and whose multiplication is not necessarily commutative. Such a structure is called a semiring, and it admits fixpoints of the form $x = a^*b$ for $x = ax + b$. A vector space where the field is replaced with a semiring is called a semimodule.

In general a semiring may admit many fixpoints, and a^*b is the least one. But to single out the least, one needs a partial order, which may be defined canonically as follows: $a \leq b$ if there is a c such that $b = a + c$. This is possible for example in $\mathbb{N}$ but not in $\mathbb{Z}$, as for any $a, b \in \mathbb{Z}$, $a + (-a + b) = b$. Hence, a semiring

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 143–153, 2011.

cannot have both a canonical order and an inverse. This is where classic and discrete mathematics diverge [4]. So what else does (not) hold in both settings?

In [5] we have shown that minimization of nondeterministic finite automata (NFA) can be advantageously cast as reachability and observability reductions of linear systems. This also allowed us to show that minimal NFA are linear transformations of each other. This result is noteworthy, as no reasonable way of relating minimal NFA was previously known (see for example [1]).

In this paper we continue the above line of work, by investigating the classic Cayley Hamilton theorem (CHT), in the context of noncommutative semirings. The class of regular languages, with NFA as their recognizers, is an important and strongly motivating subclass of these semirings.

For a matrix A with entries in a field, CHT states that A satisfies its own characteristic polynomial, that is $\mathrm{cp}_A(A)=0$ where $\mathrm{cp}_A(s)=\det(sI-A)$. Hence, any extension of CHT to noncommutative semirings has to solve two orthogonal problems: 1) The lack of subtraction; and 2) The lack of commutativity. They are both critical ingredients in the computation of the determinant $\det(sI-A)$.

The lack of subtraction was addressed in 1964 by Rutherford [10]. The main idea is to define subtraction in terms of addition by replacing terms $a-b$ with pairs (a,b). Consequently, $\det(sI-A)$ becomes $(\det^+(sI-A),\det^-(sI-A))$, a bideterminant, and CHT becomes $\mathrm{cp}^+_A(A)=\mathrm{cp}^-_A(A)$. This allowed Rutherford to extend CHT to matrices with entries in a commutative semiring.

The lack of commutativity is addressed in this paper. The main idea is to define a commutative multiplication in terms of multiplication and addition by replacing products ab with their permutation closure $[\![\, ab \,]\!]=ab+ba$. Consequently, $\det(sI-A)$ becomes $[\![\, \det(sI-A) \,]\!]$, what we call a pideterminant, and CHT becomes $[\![\, \mathrm{cp}_A(A) \,]\!]=0$. This allows us to extend CHT to any noncommutative structure, and in particular to noncommutative rings.

Combining Rutherford's solution with our own solution, allows us to extend CHT to noncommutative semirings as $[\![\, \mathrm{cp}^+_A(A) \,]\!]=[\![\, \mathrm{cp}^-_A(A) \,]\!]$. We argue that both solutions are also canonical, in the sense that $\det(A)=\det^+(A)-\det^-(A)$ and that $\det(A)=(1/n!)[\![\, \det(A) \,]\!]$ in any field.

Interpreting matrix A as a process, we also observe that the power A^n occurring in CHT can be understood in two ways: 1) As n copies of process A that interleave a single move; 2) As one copy of process A that performs n moves. This observation gives a computational justification for permutation closure, and paves the way to two different forms of CHT. Finally, considering addition idempotent, as in the class of languages, leads to a simpler form of CHT.

The rest of the paper is organized as follows. Sections 2 and 3 review semirings, fields and permutations. Section 4 reviews (bi)determinants, and introduces our new notion of pideterminant. Section 5 follows the same pattern for characteristic (bi)polynomials and pipolynomials. Sections 6 and 7 extend CHT by allowing or disallowing interleaving, respectively, and prove the validity of these extensions in noncommutative semirings. Section 8 particularizes the second version of CHT to idempotent noncommutative semirings. Finally, Section 9 contains our concluding remarks and directions for future work.

2 Semirings and Fields

A *semiring* $\mathbb{S} = (S, +, \times)$ is a set S possessing two binary operations, *addition* and *multiplication* together with their *identities* 0 and 1, respectively. Addition is required to be *associative and commutative* and multiplication is required to be *associative*. Multiplication *distributes* over addition and 0 is an *absorbing* element for multiplication, that is, for all $a \in S$, $a \times 0 = 0 \times a = 0$.

A semiring where multiplication is commutative is called a *commutative semiring*, and a semiring where addition is idempotent (for all $a \in S$, $a + a = a$) is called an *idempotent semiring*. For example, the natural numbers $\mathbb{N}$ with the usual meaning of $+$ and $\times$, form a commutative semiring.

The class of *languages* over a finite alphabet A is an idempotent semiring $\mathbb{A} = (\wp(A^*), +, \times)$, where the elements in A^* are *words* (sequences) over A, the elements in $\wp(A^*)$ are sets of words (languages), $+$ is interpreted as *union* and $\times$ is interpreted as *concatenation*. A language is *regular*, if it is accepted by a finite automaton. For example, $\{a\}$ and $\{b\}\sum_{n\in\mathbb{N}}\{a\}^n$ are regular languages.

A *field* $\mathbb{F} = (F, +, \times)$ is a commutative semiring where $+$ and $\times$ have inverses. For example, $\mathbb{R}$ and $\mathbb{C}$ are fields, with the usual meaning of $+$ and $\times$.

3 Permutations

A *permutation* π is a bijection of the finite set $\{1,\ldots,n\}$ onto itself. It has associated a *directed graph* $G(\pi) = (V, E)$, with set of vertices $V = \{1 \ldots n\}$, and set of edges E, consisting of pairs $(i, \pi(i))$, one for each for $i \in V$. For example:

$$\pi = \{(1,2),(2,3),(3,4),(4,1),(5,7),(6,6),(7,5)\}$$

is a permutation of the set $\{1,\ldots,7\}$. Its associated graph $G(\pi)$, which is shown in Figure 1, illustrates an important property of the graph of any permutation: it decomposes into *elementary disjoint cycles*, the *partial rotations* of π.

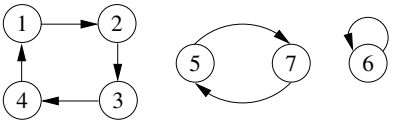


Fig. 1. The graph $G(\pi)$ of permutation π

If $G(\pi)$ has an even (odd) number of cycles with even number of edges, then π is said to have positive (negative) *sign*. For example, the graph $G(\pi)$ above, has two cycles of even length, so π has positive sign. Denote by $P(n)$ be the set of all permutations of the set $\{1 \ldots n\}$, and by $P^+(n)$ and $P^-(n)$, its positive and negative subsets, respectively.

A *partial permutation* π of the finite set $V = \{1,\ldots,n\}$ is a permutation of a subset S of V. For example, if $V = \{1 \ldots 7\}$ and π is defined as follows:

$$\pi = \{(1,2),(2,1),(3,4),(4,6),(6,3)\}$$

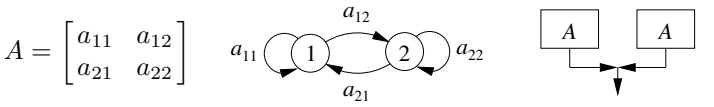


Fig. 2. (a) Matrix A. (b) $G(A)$. (c) $[\![A^2]\!]$.

then π is a permutation of its domain $\mathrm{dom}(\pi) = \{1, 2, 3, 4, 6\}$, and a partial permutation of V. Every partial permutation π of V can be extended to a permutation $\hat{\pi}$ of V by letting $\hat{\pi}(i) = \pi(i)$ if $i \in \mathrm{dom}(\pi)$ and $\hat{\pi}(i) = i$, otherwise.

Given a permutation $\pi \in P(n)$, we write $\overline{\pi}$ for the sequence $(1, \pi(1)) \ldots (n, \pi(n))$. We extend permutations $\pi \in P(n)$ to sequences $w = w_1 \ldots w_n$ in a componentwise fashion: $\pi(w) = w_{\pi(1)} \ldots w_{\pi(n)}$. For example, if $w = abcdefg$ and π is the permutation shown in Figure 1, then $\pi(w) = dabcgfe$. Similarly, if σ is another permutation of the set $\{1, \ldots 7\}$ then $\pi(\overline{\sigma})$ is equal to:

$$(4, \sigma(4))(1, \sigma(1))(2, \sigma(2))(3, \sigma(3))(7, \sigma(7))(6, \sigma(6))(5, \sigma(5))$$

4 The Determinant in Noncommutative Semirings

A square matrix A of order n with entries in a field $\mathbb{F}$ is an element of $F^{n \times n}$. One says that A has n rows and n columns. For example, a matrix A of order 2 (a 2 by 2 matrix) is shown in Figure 2(a). Row 1 is (a_{11}, a_{12}), row 2 is (a_{21}, a_{22}), column 1 is (a_{11}, a_{21}) and column 2 is (a_{21}, a_{22}).

A square matrix A of order n has associated a *weighted directed graph* $G(A) = (V, E, A)$, where $V = \{1, \ldots, n\}$ is the set of *vertices* and $E = \{(i, j) \in V^2 \mid A_{ij} \neq 0\}$ is the set of *edges* (i, j) with *weight* A_{ij}. For example, the graph $G(A)$ of the above matrix A of order 2 is shown in Figure 2(b).

A *generalized path* p in $G(A)$ is a sequence of edges $p_1 \ldots p_n$ in E. This is called a *path* if $\mathrm{head}(p_i) = \mathrm{tail}(p_{i+1})$ for each $i < n$.[1] The product $p(A) = A_{p_1} \ldots A_{p_n}$ is called the *weight of* p. For example, $(1, 1)(1, 2)(2, 1)(A) = A_{11} A_{12} A_{21}$. A path that starts and ends with same vertex is called a *cycle*. This is called *simple* if it has no other repeated vertices.

Using permutations, generalized paths and associated path weights, one can explicitly define the *determinant* of a square matrix A of order n, as follows:

$$\det(A) \;=\; \left(\textstyle\sum_{\pi \in P^+(n)} \overline{\pi} \;-\; \sum_{\pi \in P^-(n)} \overline{\pi}\right)(A)$$

where each term of det is applied to A. We denote by $\det^+$ and $\det^-$ the positive and the negative parts of the determinant operator det, respectively.

Since the determinant is an n-linear function, its value is typically computed iteratively, by expanding it along one of the rows (or columns) of its argument matrix, and then repeating the process for each remaining submatrix, until the argument matrix has only one entry (Laplace expansion).

[1] This definition of paths is more convenient in our setting.

Rutherford has transfered the determinant's computation from a commutative semirings to a ring-like structure by defining subtraction in terms of addition of tuples. Hence, the determinant has become a tuple, called a *bideterminant*:

$$\mathrm{bdt}(A) = (\det^+,\ \det^-)(A) = (\det^+(A),\ \det^-(A))$$

The bideterminant can be computed by linear expansion, as discussed above, by pretending first that negation was available to compute $\det(A)$, and then separating the positive and the negative parts of the result.

For example, consider the matrix A of Figure 2(a). The set $P(2)$ has only two permutations, which are also rotations: π_1 and π_2:

$$\pi_1 = \{(1,1),\ (2,2)\} \in P^+(2), \quad \pi_2 = \{(1,2),\ (2,1)\} \in P^-(2)$$

Using the Laplace expansion, first for row 1 and then for row 2 of A, and first for row 2 and then for row 1, one obtains the following bideterminants:

$$\mathrm{bdt}_{12}(A) = (a_{11}a_{22},\ a_{12}a_{21}), \quad \mathrm{bdt}_{21}(A) = (a_{22}a_{11},\ a_{21}a_{12})$$

In commutative semirings $\mathrm{bdt}_{12}(A) = \mathrm{bdt}_{21}(A)$. In *noncommutative semirings*, however, this is generally not true, i.e., $\mathrm{bdt}_{12}(A) \neq \mathrm{bdt}_{21}(A)$.

For example, in regular languages, the graph $G(A)$ corresponds to a *finite automaton*, and F is a finite set, called the *input alphabet*. As a consequence, the sequence (word) of inputs $a_{11}a_{22}$ is different from $a_{22}a_{11}$, unless $a_{11} = a_{22}$.

While extensive work has been devoted to determinants of matrices with entries in noncommutative rings [3], the author is not aware of any definition of determinants for matrices with entries in noncommutative semirings. Moreover, the definitions of determinants in noncommutative rings, for example the *quasideterminants* of [3], do not have determinants as a particular case, and involve division. This operation, however, is not available in semirings.

Inspired by Rutherford, we transfer the determinant's computation to a structure possesing a commutative multiplication defined in terms of addition and multiplication. This is equivalent to extending the notion of determinant to a *pideterminant* which is the sum of all row (or column) expansions. Hence:

$$\mathrm{pdt}(A) = (a_{11}a_{22} + a_{22}a_{11},\ a_{12}a_{21} + a_{21}a_{12})$$

Note that if the semiring is commutative, $\mathrm{pdt}(A) = 2!\,\mathrm{bdt}(A)$. Let $\pi(\mathrm{bdt}_{12})$ be defined as $(\pi(\mathrm{bdt}^+_{12}), \pi(\mathrm{bdt}^-_{12}))$, the π-permutation of bdt_{12}. Then:

$$\mathrm{bdt}_{12}(A) = \pi_1(\mathrm{bdt}_{12})(A), \qquad \mathrm{bdt}_{21}(A) = \pi_2(\mathrm{bdt}_{12})(A)$$

In general, the expansion of a determinant for rows $r_1, \ldots, r_n$, in this order, results in a permutation π of $\mathrm{bdt}_{12\ldots n}$, where $\pi = \{(1,\ r_1), \ldots, (n,\ r_n)\}$. Moreover, expanding recursively all rows of a matrix results in all possible permutations, and the positive (negative) sign of the arguments is preserved.

Hence, given a matrix A of order n with entries in a noncommutative semiring, one obtains the following *explicit* representation of a pideterminant:

$$\mathrm{pdt}\,(A) \ = \ \Big(\sum_{\substack{\pi \in P^+(n)\\ \sigma \in P(n)}} \sigma(\overline{\pi}),\ \ \sum_{\substack{\pi \in P^-(n)\\ \sigma \in P(n)}} \sigma(\overline{\pi})\Big)\,(A)$$

For notational simplicity we denote the *permutation closure* $\sum_{\pi \in P(n)} \pi(w)$ of w as $[\![w]\!]$. Using this notation we can write $\mathrm{pdt} = ([\![\det^+]\!], [\![\det^-]\!])$. To further simplify the notation we will write when convenient $[\![\det^+(A)]\!]$ for $[\![\det^+]\!](A)$, and pretend that we worked in a free (permutation closed) semiring.

5 The Characteristic Polynomial in Noncomm. Semirings

The *characteristic polynomial* $\mathrm{cp}_A(s)$ in indeterminate s, associated to a matrix A of order n with entries in a field, is defined as $\det(sI - A)$. For example, for the second order matrix A of Figure 2(a) one has:

$$\mathrm{cp}_A(s) \;=\; \det(\begin{bmatrix} s - a_{11} & -a_{12} \\ -a_{21} & s - a_{22} \end{bmatrix}) \;=\; s^2 - (a_{11} + a_{22})s + (a_{11}a_{22} - a_{12}a_{21})$$

The characteristic polynomial is used to compute the eigenvalues of a matrix A with entries in a real field by finding the roots of the equation $\mathrm{cp}_A(s) = 0$. Eigenvalues and their associated eigenvectors are essential tools for computing the explicit solution of systems of linear difference and differential equations.

The characteristic polynomial was generalized by Rutherford to a *characteristic bipolynomial* $\mathrm{cbp}_A(s)$ for matrices with entries in a commutative semiring:

$$\mathrm{cbp}_A(s) = (\mathrm{cp}^+_A(s),\, \mathrm{cp}^-_A(s))$$

This polynomial can be computed by first pretending one works in a field, and then separating the positive and the negative terms. For matrix A of Figure 2(a):

$$\mathrm{cbp}_A(s) = (s^2 + a_{11}a_{22},\; (a_{11} + a_{22})s + a_{12}a_{21})$$

We define the *characteristic pipolynomial* $\mathrm{cpp}_A(s)$ of a matrix A with entries in a noncommutative semiring as follows:

$$\mathrm{cpp}_A(s) = ([\![\mathrm{cp}^+_A(s)]\!],\, [\![\mathrm{cp}^-_A(s)]\!]) = (\mathrm{cpp}^+_A(s), \mathrm{cpp}^-_A(s))$$

To compute $\mathrm{cpp}_A(s)$ one can pretend to work in a free semiring when computing the closure of $\mathrm{cbp}_A(s)$. For example, for matrix A matrix of Figure 2(a):

$$\mathrm{cpp}_A(s) = ([\![s^2 + a_{11}a_{22}]\!],\; [\![(a_{11} + a_{22})s + a_{12}a_{21}]\!])$$

6 Multi-process CHT for Noncommutative Semirings

The Cayley-Hamilton theorem (CHT) is a classic result in linear algebra over fields stating that a matrix satisfies its own characteristic polynomial: $\mathrm{cp}_A(A) = 0$.

One of the applications of CHT is to compute the dimension of the A-cyclic vector space $V_A = \{A^n \mid n \in N\}$. This vector space is fundamental in the study of observability and controllability of linear systems.

For example, if the state-space description of a linear system is given by the following difference equations:

$$x(n+1) = Ax(n) + Bu(n), \quad y(n) = Cx(n), \quad x(0) = x_0$$

and V_A has dimension k, then the observability and controllability matrices of the system are defined as follows: $O = [C\,CA \ldots CA^{k-1}]^t$, $K = [B\,AB \ldots A^{k-1}B]$.

In [5] we have shown that these matrices are also relevant in the minimization of nondeterministic finite automata (NFA). Moreover, we have proved that minimal NFA are linear transformations of each other. Relating minimal NFAs is a problem that was addressed, but not properly solved, before (see e.g. [1]).

Since $\mathrm{cp}_A(A)$ is a matrix equation, the following *conventions* are used: s is replaced with A, as is replaced with aA and every constant k is replaced with kA^0. For example, for the matrix A of Figure 2(a) one obtains:

$$\mathrm{cp}_A(A) = A^2 - (a_{11} + a_{22})A + (a_{11}a_{22} - \mathtt{a}_{12}a_{21})I = 0$$

This result has been extended to commutative semirings by Rutherford in [10], and a combinatorial proof was given later by Straubing in [11]. The generalized Cayley-Hamilton theorem states that: $\mathrm{cbp}_A^+(A) = \mathrm{cbp}_A^-(A)$.

It is easy to show that CHT does not hold in noncommutative semirings. For example, consider the matrix A of Figure 2(a). Then one would require that:

$$A^2 + a_{11}a_{22}I \;=\; (a_{11} + a_{22})A + a_{12}a_{21}I$$

Now compute the LHS and the RHS of the above equation:

$$\mathrm{LHS} = \begin{bmatrix} a_{11}a_{11} + a_{12}a_{21} + a_{11}a_{22} & a_{11}a_{12} + a_{12}a_{22} \\ a_{21}a_{11} + a_{22}a_{21} & a_{21}a_{12} + a_{22}a_{22} + a_{11}a_{22} \end{bmatrix}$$

$$\mathrm{RHS} = \begin{bmatrix} a_{11}a_{11} + a_{12}a_{21} + a_{22}a_{11} & a_{11}a_{12} + a_{22}a_{12} \\ a_{11}a_{21} + a_{22}a_{21} & a_{12}a_{21} + a_{22}a_{22} + a_{11}a_{22} \end{bmatrix}$$

They are obviously different because of the lack of commutativity of the entries in A. One of the main results of this paper is that a matrix A satisfies its own characteristic pipolynomial: $\mathrm{cpp}_A^+(A) = \mathrm{cpp}_A^-(A)$.

Theorem 1. (Multi-process CHT for noncommutative semirings) *A matrix A with entries in a noncommutative (semi)ring satisfies its own characteristic pipolynomial. That is,* $\mathrm{cpp}_A^+(A) = \mathrm{cpp}_A^-(A)$.

Consider the CHT equation of the example above. Two offending weights are $a_{11}a_{22}$ of $(\mathrm{LHS})_{11}$ and $a_{22}a_{11}$ of $(\mathrm{RHS})_{11}$. These weights may be not equal in a noncommutative semiring. However, their permutation closure is the same:

$$[\![\, a_{11}a_{22} \,]\!] = [\![\, a_{22}a_{11} \,]\!] = a_{11}a_{22} + a_{22}a_{11}$$

CHT can be intuitively understood as an *incremental construction* of $[\![\mathrm{cbp}_A^+(A)]\!]$ and $[\![\mathrm{cbp}_A^-(A)]\!]$ such that at the end $[\![\mathrm{cbp}_A^+(A)]\!] = [\![\mathrm{cbp}_A^-(A)]\!]$. The construction is justified with the help of the graph $G(A)$ associated with A.

For illustration purpose, we will use the matrix A of order 2 shown in Figure 2 with correponding CHT $[\![\, A^2 + a_{11}a_{22}I \,]\!] = [\![\, (a_{11} + a_{22})A + a_{12}a_{21}I \,]\!]$.

Start with $\mathrm{LHS}_0 = A^2$. Each entry $(A^2)_{ij}$ is a sum of weights of length 2, each weight being associated to a path from i to j in $G(A)$. For example, the path (1,2)(2,1) has associated the weight $a_{12}a_{21}$. Since $G(A)$ has only 2 vertices, all these paths must have at least one simple cycle.

Suppose we first want to add to RHS the weights of the simple cycles of length 2 in $G(A)$. For example, (1,2)(2,1) and (2,1)(1,2), with the associated weights $a_{12}a_{21}$ and $a_{21}a_{12}$, respectively. They are all contained in the *diagonal* $\mathrm{diag}(A^2)$ of A^2. However, since the diagonal of A^2 may also contain products of cycles with length less then 2, we denote by $\mathrm{diag}_s(A^2)$ the *restriction* of $\mathrm{diag}(A^2)$ to simple cycles only. Similarly, we denote by $\mathrm{trace}_s(A^2)$, *the sum* of the simple cycles in $\mathrm{diag}_s(A^2)$. Consequently, we start with: $\mathrm{RHS}_0 = \mathrm{trace}_s(A^2)I$.

All the cycle weights in $\mathrm{trace}_s(A^2)$ are permutations (in fact rotations) of the simple-cycle weights of vertex 1. There are 2 such permutations (including identity) which we denote by π_1 and π_2. Now we can write:

$$\mathrm{RHS}_0 = \sum_{i=1}^{2} \pi_i(a_{12}a_{21})I$$

Multiplying these permutations with the identity matrix I has unfortunately undesired consequences: it introduces spurious weights in each entry of RHS_0. For example entry $(\mathrm{RHS}_0)_{11}$ also contains weights such as $a_{21}a_{12}$.

To balance out spurious weights in RHS_0 we have to add the corresponding permutations of A^2 to the LHS. Hence, the LHS becomes $\mathrm{LHS}_1 = \sum_{i=1}^{2} \pi_i(A^2)$. Obviously, this introduces many more spurious weights on the LHS.

The construction now continues by adding and balancing out cycles of length 1 on the RHS, which we omit for space limitations.

Discussion. In the above construction, most of the effort is devoted to *fixing "spurious" weights*. One may therefore wonder whether such weights make any sense, and if not, whether there was a way of getting rid of them.

Consider matrix A of order n and regard $G(A)$ as a process which either acts as an acceptor or as a generator of words over a given alphabet. The power A^n can be interpreted in two distinct ways: 1) As the *interleaving of n copies of A*, each starting in an arbitrary state and performing *one move*; 2) As *one copy of A* that performs *n moves*. In each case, one can ask what is *the sum*, if counting is important, of the words accepted (or generated) by A^n?

In the interleaving interpretation of A^n, cycle weights s.a. $a_{11}a_{22}a_{33}$ make sense: the word is generated by letting the first copy of A start in vertex 1 and make one move to generate a_{11}, then the second start in vertex 2 and make one move to generate a_{22} and finally the third start in vertex 3 and make one move to generate a_{33}. Since every copy of A can make any move before or after the other copy made one move, one has to consider all permutations.

In conclusion, *Theorem 1 explicitly defines the behavior of the process resulting from the interleaving (shuffling) of n copies of process A*. For example, Figure 2(c) shows the interleaving of two copies of matrix A of order 2.

In process algebra, commutativity of inputs is equivalent with their *independence*: one obtains the same result no matter in what order one processes them. Hence, matrices with entries in a commutative semiring correspond to processes over a set of independent inputs. For general processes, independence might hold for some subsets of the input alphabet, but not for the entire alphabet. The knowledge of an independence relation over the input alphabet is still very useful, because it allows to partition matrix A into commutative blocks.

Considering only one version of the commutative products, then corresponds to the *partial-order reduction technique* used in computer-aided verification.

7 Single-Process CHT for Noncommutative Semirings

In the second interpretation of A^n in the CHT, as one copy of A performing n moves, "spurious" cycle weights such as $a_{11}a_{22}a_{33}$ or $a_{11}a_{23}a_{32}$ make no sense. We would therefore like to find a way to get rid of them.

An *acceptor algorithm* that cleans up "wrong" weights, is to: 1) First compute $\mathrm{cpp}_A(A)$ as before, and 2) Then remove all generalized path-weights in $\mathrm{cpp}_A(A)_{ij}$ that either do not correspond to paths, or they are are misplaced, that is their corresponding starting vertex is not i or their ending vertex is not j.

For example, the weight of the generalized path $(1,1)(2,2)$ is removed because it is not a path. The wight of $(1,2)(2,1)$ is removed if it appears in $\mathrm{cpp}_A(A)_{22}$.

Theorem 2. (1ST SINGLE-PROCESS CHT FOR NC SEMIRINGS) *A matrix A with entries in a noncommutative semiring satisfies its own characteristic pipolynomial, when clean up is added at the end of the permutation closure.*

One may *avoid introducing "spurious" weights by treating cycles as diagonal matrices.* If p is a cycle, then let $\langle p \rangle$ be the diagonal matrix with $\langle p \rangle_{ii} = p$ if p belongs to $(A^{|p|})_{ii}$ and $\langle p \rangle_{ii} = 0$, otherwise. If c is a sum of cycles of same length, then let $\langle c \rangle$ denotes the sum of their corresponding matrices.

A *generator algorithm* for $\mathrm{cpp}_A(A)$ can now be defined as follows: 1) Take the rotation closure of all cycles. 2) Consider cycles atomic and take the rotation closure of the entire terms. 3) Keep the cycles fixed, and take the partial-permutation closure. We denote by $(\!(\, \mathrm{cbp}_A(A) \,)\!)$ steps 1–3.

For example, let $\langle c \rangle = \langle a_{12}a_{21} + a_{21}a_{12} \rangle$ be the rotation closure of cycle matrix $\langle a_{12}a_{21} \rangle$. Then:

$$(\!(\, \langle a_{12}a_{21} \rangle A^2 \,)\!) = \begin{array}{c} \langle c \rangle A^2 \;+\; A\langle c \rangle A \;+\; A^2\langle c \rangle + \\ \pi_1(\langle c \rangle A^2) + \pi_2(A\langle c \rangle A) + \pi_3(A^2\langle c \rangle) \end{array}$$

where π_1, π_2 and π_3 swap positions 2 and 3, 1 and 3, and 1 and 2, in $\langle c \rangle A^2$ $A\langle c \rangle A$ and $A^2\langle c \rangle$, respectively.

Theorem 3. (2ND SINGLE-PROCESS CHT FOR NONCOMMUTATIVE SEMIRINGS) *In a noncommutative semiring $(\!(\, \mathrm{cbp}^+_A(A) \,)\!) = (\!(\, \mathrm{cbp}^-_A \,)\!)$ for any matrix A if each cycle c is interpreted as the (diagonal) matrix $\langle c \rangle$.*

For example, the CHT for matrix A of order 2 simplifies to:

$$(\!(\, A^2 \,)\!) = \langle a_{11} + a_{22} \rangle A + A\langle a_{11} + a_{22} \rangle + \langle a_{12}a_{21} + a_{21}a_{12} \rangle$$

8 The CHT for Noncommutative Idempotent Semirings

Suppose now that the entries of matrix A belong to a noncommutative *idempotent semiring.* Recall that a semiring S is called idempotent, if its *additive*

operation is idempotent, that is, for any element a of the semiring, $a + a = a$. Noncommutative idempotent semirings are important, as they contain as a particular case the class of *regular languages*.

Although counting is not important in such semirings, it is not obvious (at least to the author) how the multi-process CHT could be further simplified. One still needs the permutation closure $[\![\mathrm{cbp}_A(A)]\!]$, but addition simplifies.

The single-process version of the CHT can be however, further simplified in idempotent semirings. Let us denote by $(\!(\mathrm{cbp}_A(A))\!)$ the closure operation discussed in the previous section, where the last step, the partial-permutation closure, is discarded. Then we have the following theorem.

Theorem 4. (SINGLE-PROCESS CHT FOR NC IDEMPOTENT SEMIRINGS) *Let A be a matrix of order n with entries in a noncommutative idempotent semiring. If each cycle c in $\mathrm{cbp}_A(A)$ is interpreted as the (diagonal) matrix $<c_k>$, then its Cayley-Hamilton theorem simplifies to $A^n = (\!(\mathrm{cbp}^-_A(A))\!)$.*

For example, consider matrix A of Figure 2(a), and assume its entries are distinct and belong to a noncommutative idempotent semiring. Then the CHT of this matrix simplifies to the following form:

$$A^2 = <a_{11} + a_{22}>A + A<a_{11} + a_{22}> + <a_{12}a_{21} + a_{21}a_{12}>$$

9 Conclusions

We have extended the Cayley-Hamilton theorem (CHT), a classic result in linear algebra over fields which states that a matrix satisfies its own characteristic polynomial, to noncommutative semirings.

The pideterminant and the pipolynomial we have defined for this purpose could have also been inferred by using the *shuffle product* of [9] and an evaluation function *eval*. Given a noncommutative ring R one can define a commutative ring $S = (R^*, +, \|)$ where R^* consists of sequences in R and $s\|t$ is the shuffle product of $s, t \in R^*$, defined in terms of addition and concatenation.

Since S is commutative, the computation of $\det(A)$ is well defined and so is the Cayley-Hamilton theorem. As a consequence, $\mathrm{pdt}(A) = eval(\det(A))$ where *eval* replaces concatenation with the product in R and evaluates the result.

In S the power A^n is the n-shuffle (interleaving) of A, and it can be expressed as a linear combination of $I, A, \ldots A^{n-1}$. This observation suggests a generalization of linear dependence for a set of vectors $x_1, \ldots, x_n$ in a noncommutative module as follows: there are scalars $a_1, \ldots, a_n$ such that the shuffle product $a_1\|x_1 + \ldots + a_n\|x_n = 0$. Such extensions are the focus of future work.

References

1. Nivat, M., Arnold, A., Dicky, A.: A note about minimal nondeterministic automata. EATCS 45, 166–169 (1992)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicolin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Comp. Sci. 138, 3–34 (1995)

3. Gelfand, I., Gelfand, S., Retakh, V., Wilson, R.L.: Quasideterminants. Adv. Math. 193, 1–82 (2005)
4. Gondran, M., Minoux, M.: Graphs, Dioids and Semirings. Springer, Heidelberg (2008)
5. Grosu, R.: Finite automata as time-invariant linear systems: Observability, reachability and more. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 194–208. Springer, Heidelberg (2009)
6. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages POPL 1973, pp. 194–206. ACM, New York (1973)
7. Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O automata. Inf. and Comp. 185(1), 103–157 (2003)
8. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, New York (1992)
9. Ree, R.: Lie elements and an algebra associated with shuffles. Anals of Mathematics 67(2), 210–220 (1958)
10. Rutherford, D.E.: The Cayley-Hamilton theorem for semi-rings. Proc. Roy. Soc. Edinburgh Sect. A 66, 211–215 (1964)
11. Straubing, H.: A combinatorial proof of the Cayley-Hamilton theorem. Discrete Maths. 43, 273–279 (1983)

Approximating Minimum Reset Sequences

Michael Gerbush[1] and Brent Heeringa[2]

[1] Department of Computer Science, The University of Texas at Austin, Taylor Hall 2.124, Austin, TX 78712
mgerbush@cs.utexas.edu
[2] Department of Computer Science, Williams College, 47 Lab Campus Drive, Williamstown, MA 01267
heeringa@cs.williams.edu

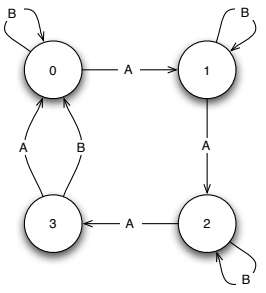
Abstract. We consider the problem of finding minimum reset sequences in synchronizing automata. The well-known Černý conjecture states that *every n-state synchronizing automaton has a reset sequence with length at most* $(n-1)^2$. While this conjecture gives an upper bound on the length of every reset sequence, it does not directly address the problem of finding the *shortest* reset sequence. We call this the MINIMUM RESET SEQUENCE (MRS) problem. We give an $O(kmn^k + n^4/k)$-time $\lceil \frac{n-1}{k-1} \rceil$-approximation for the MRS problem for any $k \geq 2$. We also show that our analysis is tight. When $k = 2$ our algorithm reduces to Eppstein's algorithm and yields an $(n-1)$-approximation. When $k = n$ our algorithm is the familiar exponential-time, exact algorithm. We define a nontrivial class of MRS which we call STACK COVER. We show that STACK COVER naturally generalizes two classic optimization problems: MIN SET COVER and SHORTEST COMMON SUPERSEQUENCE. Both these problems are known to be hard to approximate, although at present, SET COVER has a slightly stronger lower bound. In particular, it is **NP**-hard to approximate SET COVER to within a factor of $c \cdot \log n$ for some $c > 0$. Thus, the MINIMUM RESET SEQUENCE problem is as least as hard to approximate as SET COVER. This improves the previous best lower bound which showed that it was **NP**-hard to approximate the MRS on binary alphabets to within any constant factor. Our result requires an alphabet of arbitrary size.

1 Introduction

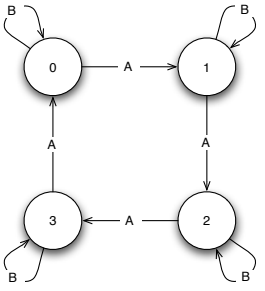
In the *part orienteering problem* [1], a part drops onto a pan which is moving along a conveyor belt. The part is in some unknown orientation. The goal is to move the part into a known orientation through a sequence of pan tilts. The sequence of tilts should be *universal* in the sense that, regardless of its initial position, the tilts always bring the part back to the some known orientation. If Q is a finite set of n possible states that the part can occupy, Σ is a finite alphabet of m symbols representing the types of tilts, and $\delta : Q \times \Sigma \to Q$ is a state transition function mapping states to states based on the action of a tilt from Σ, then $A = (Q, \Sigma, \delta)$ forms a simple deterministic finite automaton (omitting start and accept states).

We extend δ to sequences $\delta : Q \times \Sigma^* \to Q$ in the natural way: $\delta(q, \varepsilon) = q$ and $\delta(q, xw) = \delta(\delta(q, x), w)$ where $q \in Q$, ε is the empty sequence, $x \in \Sigma$ and $w \in \Sigma^*$.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 154–162, 2011.



(a) A synchronizing automaton



(b) A non-synchronizing automaton

Fig. 1. An example of a synchronizing and a non-synchronizing automata. Notice that the automaton in (a) admits the reset sequence BAAABAAAB while the automaton in (b) has no reset sequence.

We also let δ operate on *sets of states* instead of a single state. That is, if $Q' \subseteq Q$ and $z \in \Sigma^*$ then $\delta(Q', z) = \{\delta(q, z) \mid q \in Q'\}$.

Given an automaton $A = (Q, \Sigma, \delta)$, a sequence $w \in \Sigma^*$ is a *reset sequence* for A if and only if $|\delta(Q, w)| = 1$. We call w a reset sequence because it *resets* the state of a finite automaton back to some known state. Given the automaton formulation of the part orienteering problem above, it is easy to see that finding an orienting sequence of tilts is equivalent to finding a reset sequence in an automaton.

Some automata have reset sequences and some do not (see Figure 1). If an automaton has a reset sequence, then we say it is *synchronizing*. Checking if an automaton is synchronizing can be done in polynomial time using dynamic programming [2], however finding the shortest reset sequence is **NP**-hard [3]. We call finding the shortest reset sequence of an automaton the MINIMUM RESET SEQUENCE (MRS) problem. Since it is unlikely that efficient algorithms exist for the MRS problem, we consider approximating the shortest reset sequence.

1.1 Prior and Related Work

Arguably the most well-known open problem in automata theory today is a conjecture on the length of the minimum reset sequence. Posed in 1964, Černý conjectured that any n-state, synchronizing automaton has a reset sequence with length at most $(n-1)^2$ [4]. Over 40 years later, this problem remains open and continues to draw considerable attention. The current best upper bound on the MRS is $(n^3 - n)/6$ [5,6]. For large classes of automata, Černý's conjecture holds [3,7,8]. In particular, Eppstein [3] gives a polynomial time algorithm for finding reset sequences with length at most $(n-1)^2$ for automata where the transition function has certain monotonicity properties and Kari [8] shows that Černý's conjecture holds when the graph underlying the automaton is Eulerian.

While Černý's conjecture claims an upper bound on the length of the shortest reset sequence, it does not directly address the problem of *finding* the shortest reset sequence. For example, in the part orienteering problem, when an $O(\log n)$ reset sequence exists,

improving from an $O(n^2)$ solution to an $O(n \log n)$ solution may provide an enormous savings in production costs. Berlinkov recently showed that it **NP**-hard to approximate the MRS problem to within any constant factor [9]. This result holds for binary alphabets. In addition, Olschewski and Ummels showed that finding the minimum reset sequence and determining the length of the minimum reset sequence are in $\mathbf{FP}^{\mathbf{NP}}$ and $\mathbf{FP}^{\mathbf{NP}[log]}$ respectively [10].

1.2 Results

We begin in Section 2 by giving a simple $O(kmn^k + n^4/k)$-time $\lceil \frac{n-1}{k-1} \rceil$-approximation for MINIMUM RESET SEQUENCE for any $k \geq 2$. Here n is the number of states in the automaton and m is the size of the alphabet. When $k = 2$, this algorithm reduces to Eppstein's algorithm [3] so our analysis shows that his algorithm produces an $(n-1)$-approximation. When $k = n$ our algorithm becomes the standard exponential-time, exact algorithm. We also show that our analysis is tight. In Section 3 we define a non-trivial class of MRS which we call STACK COVER. We show that STACK COVER is a natural generalization of two classic optimization problems: SET COVER and SHORTEST COMMON SUPERSEQUENCE. Under the assumption that $\mathbf{P} \neq \mathbf{NP}$, SHORTEST COMMON SUPERSEQUENCE has no α-approximation for any constant $\alpha > 0$ [11]. This matches the lower bound given by Berlinkov, albeit for alphabets of arbitrary, instead of constant size. However, assuming $\mathbf{P} \neq \mathbf{NP}$, SET COVER has no $c \cdot \log n$-approximation for some constant $c > 0$ [12]. This offers a significant improvement over the previous best lower bound. We conclude in Section 4 with some conjectures and open problems. In particular, we offer a roadmap for combining the hardness of approximating SHORTEST COMMON SUPERSEQUENCE and SET COVER to achieve even stronger lower bounds for MINIMUM RESET SEQUENCE.

2 A Simple Approximation Algorithm

Our algorithm relies on the following property.

Property 1. Let $A = (Q, \Sigma, \delta)$ be a finite automaton. For every $k \geq 2$, A is synchronizing if and only if for every $Q' \subseteq Q$ such that $|Q'| \leq k$ there exists $x \in \Sigma^*$ such that $|\delta(Q', x)| = 1$.

Proof. When $k = 2$, the property reduces to a well-known necessary and sufficient condition for synchronization [2]. It is easy to see that since the property holds for $k = 2$ then it holds in general since any synchronizing sequence $x \in \Sigma^*$ for $\{\delta(p, y), \delta(s, y)\} \subseteq Q$ can be appended to a synchronizing sequence $y \in \Sigma^*$ for $\{p, q\} \subseteq Q$ to form a synchronizing sequence yx for $\{p, q, s\}$. □

Given some $k \geq 2$ we create a k-dimensional dynamic programming table D such that, for all subsets $Q' \subseteq Q$ where $2 \leq |Q'| \leq k$ if $x = ay$ is an MRS for Q' where $a \in \Sigma$ and $y \in \Sigma^*$ then $D(Q') = (a, \delta(Q', a))$. That is, D yields the first letter in the MRS for Q' as well as pointer to the next subset of states in the MRS. The base case is when $|Q'| = 1$. In this case we simply return the empty sequence. The following lemma establishes an upper bound on the time required to construct D.

Lemma 1. *Given $k \geq 2$, constructing D takes times $O(mn^k)$.*

Proof. Given an automaton $A = (Q, \Sigma, \delta)$, define an edge-labelled, directed multigraph $G = (V, E)$ such that $V = \{Q' \subseteq Q \mid 1 \leq |Q'| \leq k\}$ and for every $U, W \in V$ we have $(U, W) \in E$ labelled with $a \in \Sigma$ if and only if $\delta(W, a) = U$. That is, if a brings W to U then there is an edge from U to W labelled with a. We perform a breadth-first search on G, starting with all the singleton sets of Q. That is, we begin by considering all sets Q' such that $|Q'| = 1$. Whenever we encounter a node $R \in V$ for the first time we let $D(R) = (a, R')$ where R' is the parent of R in the breadth-first search and (R', R) is labelled with a. We let $D(Q') = \varepsilon$ if $|Q'| = 1$. If the breadth-first search fails to reach every node in V then, by Property 1, A is not synchronizing. Correctness of the construction follows from the fact that we are always considering the shortest sequences that take a singleton node to a non-singleton node (and, by way of reversing the edge orientations, the shortest reset sequences). Since the graph has $O(n^k)$ nodes and $O(mn^k)$ edges and we are performing a simple BFS, constructing the table takes time $O(mn^k)$. □

Algorithm 1. APPROX-MRS(A, k) where $A = (Q, \Sigma, \delta)$

```
1:  X ← Q
2:  z ← ϵ
3:  Let D be the dynamic programming table given by Lemma 1.
4:  while |X| > 1 do
5:      α ← min{|X|, k}
6:      Choose an arbitrary subset Q' ⊆ X such that |Q'| = α.
7:      while |Q'| > 1 do
8:          (a, Q') ← D(Q')
9:          z ← z · a
10:         X ← δ(X, a)
11:     end while
12: end while
13: return z
```

Algorithm 1 uses D to successively synchronize k states until only one state remains. The correctness of the algorithm follows from Property 1.

Theorem 1. *Algorithm 1 is an $O(kmn^k + n^4/k)$-time $\lceil \frac{n-1}{k-1} \rceil$-approximation for the* MINIMUM RESET SEQUENCE *problem for any $k \geq 2$.*

Proof. Let $k \geq 2$ be given and let z be the MRS for A with length OPT. For any $Q' \subseteq Q$ such that $|Q'| \leq k$, if y is the sequence given for Q' by D then $|y| \leq OPT$. This follows by construction since D gives us a method to find the shortest reset sequence for all $Q' \subseteq Q$ such that $|Q'| \leq k$. If $OPT < |y|$ then z would be a shorter synchronizing sequence for Q', a contradiction.

Notice that in each iteration of line 4 (other than the final iteration) we decrease the size of X by $k - 1$ (once $|X| < k$ we perform at most one additional iteration). Thus, after at most $\lceil \frac{n-1}{k-1} \rceil$ iterations X contains a single state. Since each iteration of line 4

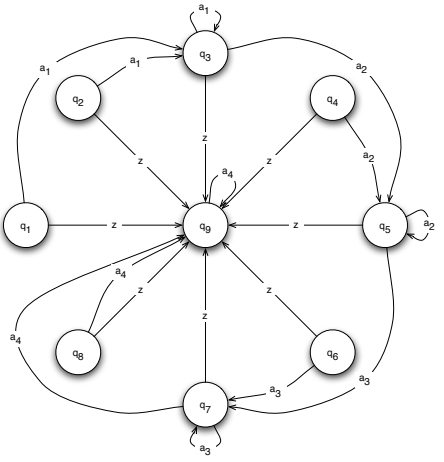


Fig. 2. A tight example for the case $n = 9$ and $k = 3$. All transitions not shown are self-transitions. The optimal solution is z, but Algorithm 1 could return the solution $a_1a_2a_3a_4$.

appends a sequence of length at most OPT to z, Algorithm 1 yields a sequence with length at most $\lceil \frac{n-1}{k-1} \rceil \cdot OPT$ which yields the desired approximation ratio.

Constructing the dynamic programming table takes time $O(kmn^k)$. The inner loop in line 7 executes at most $O(n^4/k)$ times since $O(n^3)$ is an upper bound on the reset sequence. Since the outer loop in line 4 executes at most n times, Algorithm 1 takes time at most $O(kmn^k + n^4/k)$. □

When $k = 2$, Algorithm 1 reduces to Eppstein's algorithm. Thus, this algorithm yields an $(n-1)$-approximation for the MINIMUM RESET SEQUENCE problem. When $k = n$ the dynamic programming table becomes the power-set automata and a breadth-first search yields the standard exponential-time exact algorithm [13]. It also turns out that the analysis of Algorithm 1 is tight.

Theorem 2. *The analysis of Algorithm 1 is tight.*

Proof. We construct an instance of the MRS problem such that Algorithm 1 gives a solution that is $\lceil \frac{n-1}{k-1} \rceil$ times larger than the optimal solution. Let n and k be given. Choose $j = \lceil \frac{n-1}{k-1} \rceil$ and take $Q = \{q_1, q_2, ..., q_n\}$ as the set of states and $\Sigma = \{z, a_1, a_2, ..., a_j\}$ as the alphabet. We define the transition function δ as

$$\delta(q_i, a_h) = \begin{cases} q_{\lceil \frac{i}{k-1} \rceil (k-1)+1} & \text{if } h = \lceil \frac{i}{k-1} \rceil \\ q_n & \text{if } a_h = z \\ q_i & \text{otherwise.} \end{cases}$$

The optimal solution to an instance of this form is the string z which has length 1. However, Algorithm 1 chooses to merge k arbitrary states. If the sequence of k-states are $(q_1, \ldots, q_k), (q_k, \ldots, q_{2(k-1)+1}), (q_{2(k-1)+1}, \ldots, q_{3(k-1)+1})$...,$(q_{(j-1)(k-1)+1}, \ldots, q_n)$,

and if for each tuple the algorithm chooses a_i instead of z to merge the states, then only k states are merged at a time. In this case, the solution that Algorithm 1 gives is $a_1 a_2 \dots a_j$ which has length $j = \lceil \frac{n-1}{k-1} \rceil$. Thus, our analysis is tight. Figure 2 gives an example for $n = 9$ and $k = 3$.

3 The STACK COVER Problem

Here we define a non-trivial class of MINIMUM RESET SEQUENCE problems which we call STACK COVER. Our primary purpose in introducing this class is to show that MINIMUM RESET SEQUENCE is hard to approximate, however, STACK COVER may have independent appeal.

Imagine you have n stacks of cards where each card is painted with multiple colors. You can peek at any card in any stack at any time. Selecting a *color* removes the top card of each stack provided that card is painted with the selected color. The goal is to select the shortest sequence of colors that empties all the stacks. This is the STACK COVER problem. When each stack has a single card, then the colors represent sets and STACK COVER becomes the SET COVER problem. When the stacks have varying heights but each card is painted with a single color then a stack of cards is a string and STACK COVER becomes the SHORTEST COMMON SUPERSEQUENCE problem. Below we review the definitions of SET COVER and SHORTEST COMMON SUPERSEQUENCE and formally show how STACK COVER generalizes both problems.

We now define STACK COVER within the MINIMUM RESET SEQUENCE framework. We treat the symbols in Σ as colors and then impose some structure on the transition function. Figure 3 shows the general form of STACK COVER instances. We partition the set of states Q into n stacks $Q_1 \cup Q_2 \cup \dots \cup Q_n$ plus a single sink state $\hat{q}$. Furthermore, we linearly order each stack Q_i as $q_{i1}, q_{i2}, \dots, q_{il_i}, q_{i(l_i+1)}$ where each q_{ij} is a state in Q_i, and for convenience, $q_{i(l_i+1)} = \hat{q}$. That is, we assume the final state in every stack is the sink state. The transition function must obey this order, so for each $1 \le i \le n$ and for each $x \in \Sigma$, either $\delta(q_{ij}, x) = q_{ij}$ or $\delta(q_{ij}, x) = q_{i(j+1)}$ for all $1 \le j \le l_i$. Finally, we have $\delta(\hat{q}, x) = \hat{q}$ for all $x \in \Sigma$. If $A = (Q, \Sigma, \delta)$ is a STACK COVER instance, then let OPT(A) be the length of the MINIMUM RESET SEQUENCE for A.

SHORTEST COMMON SUPERSEQUENCE as STACK COVER. An instance of SHORTEST COMMON SUPERSEQUENCE (SCS) is a set of strings $R = \{t_1, \dots, t_n\}$ over an alphabet Σ of size m. That is, each t_i is a string in Σ^*. The goal is to find the shortest string $w \in \Sigma^*$ such that each string t_i is a subsequence of w. We can reduce an SCS instance to a STACK COVER instance as follows. Refer to the jth character in the string t_i as t_{ij}. Given a set of strings R, construct $A = (Q, \Sigma, \delta)$, such that

$$Q = \{q_{ij} \mid 1 \le i \le n, 1 \le j \le |t_i|\} \cup \{\hat{q}\}$$

and for all $q_{ij} \in Q \setminus \{\hat{q}\}$ and $a \in \Sigma$

$$\delta(q_{ij}, a) = \begin{cases} \hat{q} & \text{if } a = t_{ij} \text{ and } j = |t_i| \\ q_{i(j+1)} & \text{if } a = t_{ij} \text{ and } j < |t_i| \\ q_{ij} & \text{otherwise.} \end{cases}$$

and $\delta(\hat{q}, a) = \hat{q}$ for all $a \in \Sigma$.

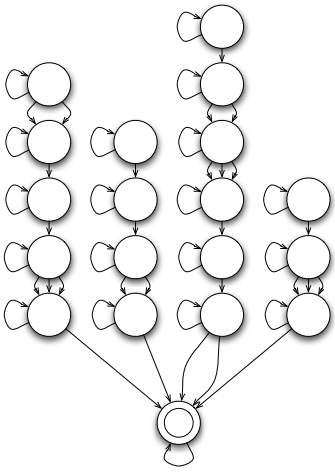


Fig. 3. An instance of STACK COVER

From the definition, notice that we have created a single state for each character in R and a transition symbol for each character in Σ. Also, we have added one additional state $\hat{q}$, which acts as a single sink node attached to each stack. Every transition from $\hat{q}$ is a self transition. The transition function δ guarantees that each state has only a single transition symbol to the next state in the stack. Notice that $w \in \Sigma^*$ is a reset sequence for A if and only if w is a supersequence for R.

Jiang and Li [11] showed that SCS has no α-approximation for any $\alpha > 0$ unless $\mathbf{P} = \mathbf{NP}$. Here n is the number of strings. This hardness result holds even when the strings have constant length, so given the reduction above, it applies to the MINIMUM RESET SEQUENCE problem where n is the number of states.

SET COVER as STACK COVER. An instance of SET COVER is a base set $X = \{x_1, \ldots, x_n\}$ and a collection $S = \{S_1, \ldots, S_m\}$ of subsets of X. The goal is to find the smallest subset $S' \subseteq S$ such that $\cup_{S_i \in S'} S_i = X$. We can reduce SET COVER to STACK COVER as follows. Given a SET COVER instance (X, S), construct $A = (Q, \Sigma, \delta)$ such that

$$Q = X \cup \{\hat{q}\}$$
$$\Sigma = \{1, \ldots, m\}$$
$$\delta(x, j) = \begin{cases} \hat{q} & \text{if } x \in S_j \\ x & \text{otherwise.} \end{cases}$$

In our automaton, we have a single state for each element of X and a single transition symbol for each subset in S. Again, we add a sink node, $\hat{q}$, that is connected to every stack and has only self transitions. We define the transition function so that a node can be brought to $\hat{q}$ if and only if a subset containing that character is selected. Notice that $w_1 \cdots w_l \in \Sigma^*$ is a reset sequence for A if and only if $\bigcup_{1 \leq i \leq l} S_{w_i}$ is a cover for X.

SET COVER has no $c \log n$-approximation for some constant $c > 0$ unless $\mathbf{P} = \mathbf{NP}$ [12]. Thus, this lower bound also extends to STACK COVER.

4 Open Problems and Conjectures

The lowers bounds in Section 3 require alphabets of finite, yet arbitrary size. It is an open problem to determine if these results extend to the case where the alphabet has constant size.

In addition, the gap between the upper bound on the approximation ratio offered by Algorithm 1 and the lower bound offered by SET COVER is quite large. One line of attack in closing this gap is to combine an instance of SHORTEST COMMON SUPERSEQUENCE with an instance of SET COVER to produce an instance of STACK COVER that is harder to approximate than either problem on its own. For example, given $A = (Q_A, \Sigma_A, \delta_A)$ and $B = (Q_B, \Sigma_B, \delta_B)$, where A represents an SCS problem and B represents a SET COVER problem, we define the natural cross product $A \times B =$

$$
\begin{aligned}
Q &= (Q_A \setminus \{\hat{q_A}\}) \times (Q_B \setminus \{\hat{q_B}\}) \cup \{\hat{q}\} \\
\Sigma &= \Sigma_A \times \Sigma_B \\
\delta((q_{ij}, q), (a, s)) &= \begin{cases} \hat{q} & \text{if } a = t_{ij} \text{ and } j = |t_i| \text{ and } \delta_B(q, s) = \hat{q_B} \\ (q_{i(j+1)}, q) & \text{if } a = t_{ij} \text{ and } j < |t_i| \text{ and } \delta_B(q, s) = \hat{q_B} \\ (q_{ij}, q) & \text{if } a \neq t_{ij} \text{ or } \delta_B(q, s) \neq \hat{q_B} \end{cases}
\end{aligned}
$$

where $\hat{q_A}$ is the sink state of A and $\hat{q_B}$ is the sink state of B.

Each state in the SCS automaton A is now paired with a state from the SET COVER automaton B, creating n stacks for each stack in the original SCS instance, where n is the number of elements in the SET COVER instance. Likewise, each transition in the SCS automaton is now paired with a transition from the SET COVER automaton, creating m transitions for each transition in the original SCS instance. Here m is the number of subsets in the SET COVER instance. The transition function has become more complex, but the general concept is straightforward: we can only move downward in a stack if we select a symbol that corresponds to both the current node's symbol in the SCS instance and to one of its subsets in the SET COVER instance.

Assuming $\text{OPT}(\cdot)$ gives the length of an optimal solution, it's clear that $\text{OPT}(A \times B) \leq \text{OPT}(A) \cdot \text{OPT}(B)$. However, if we can show that $\text{OPT}(A) \cdot \text{OPT}(B) \leq \tau \cdot \text{OPT}(A \times B)$ for some constant τ then the following conjecture holds:

Conjecture 1. For any $\alpha > 0$, the MINIMUM RESET SEQUENCE problem has no polynomial-time algorithm with approximation ratio $\alpha \log n$, where n is the total number of states, unless $\mathbf{P} = \mathbf{NP}$.

This lower bound is stronger than the lower bounds for both SET COVER and SHORTEST COMMON SUPERSEQUENCE. However, showing that $\text{OPT}(A) \cdot \text{OPT}(B) \leq \tau \cdot \text{OPT}(A \times B)$ for some constant τ seems challenging because of the interaction between A and B. More specifically, it is tempting to think that $\text{OPT}(A) \cdot \text{OPT}(B) = \text{OPT}(A \times B)$, but this is not the case. Consider an SCS instance $A = \{ab, ba\}$ and a SET COVER instance $B = (X, \mathcal{C})$ where $X = \{1, 2, 3, 4\}$ and $\mathcal{C} = \{\{2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}\}$. An optimal solution to $A \times B$ uses only fives symbols:

$$(B, \{1, 2, 4\}), (A, \{2, 3\}), (B, \{1, 3, 4\}), (A, \{1, 3, 4\}), (B, \{1, 2, 4\})$$

however OPT$(A) = 3$ (either aba or bab) and OPT$(B) = 2$ (since no subset contains all 4 elements).

Acknowledgements

The authors wish to thank the anonymous reviewers for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant No. 0812514.

References

1. Natarajan, B.K.: An algorithmic approach to the automated design of parts orienters. In: FOCS, pp. 132–142 (1986)
2. Salomaa, A.: Generation of constants and synchronization of finite automata. J. UCS 8(2), 332–347 (2002)
3. Eppstein, D.: Reset sequences for monotonic automata. SIAM J. Computing 19(3), 500–510 (1990)
4. Černý, J.: Poznamka k homogenym eksperimentom s konechnymi automatami. Math.-Fyz. Čas 14, 208–215 (1964)
5. Pin, J.E.: On two combinatorial problems arising from automata theory. Annals of Discrete Mathematics 17, 535–548 (1983)
6. Klyachko, A.A., Rystsov, I.K., Spivak, M.A.: In extremal combinatorial problem associated with the length of a synchronizing word in an automaton. Cybernetics and Systems Analysis 23(2), 165–171 (1987)
7. Ananichev, D., Volkov, M.: Synchronizing generalized monotonic automata. Theoretical Computer Science 330(1), 3–13 (2005)
8. Kari, J.: Synchronizing finite automata on eulerian digraphs. Theoretical Computer Science 295(1-3), 223–232 (2003)
9. Berlinkov, M.V.: On calculating length of minimal synchronizing words. In: Ablayev, F., Mayr, E.W. (eds.) CSR 2010. LNCS, vol. 6072, Springer, Heidelberg (2010); CoRR abs/0909.3787
10. Olschewski, J., Ummels, M.: The Complexity of Finding Reset Words in Finite Automata. CoRR abs/1004.3246v1 (April 2010)
11. Jiang, T., Li, M.: On the approximation of shortest common supersequences and longest common subsequences. SIAM J. Comput. 24(5), 1122–1139 (1995)
12. Alon, N., Moshkovitz, D., Safra, S.: Algorithmic construction of sets for k-restrictions. ACM Trans. Algorithms 2, 153–177 (2006)
13. Volkov, M.V.: Synchronizing automata and the Černý conjecture. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 11–27. Springer, Heidelberg (2008)

Transductions Computed by PC-Systems of Monotone Deterministic Restarting Automata

Norbert Hundeshagen, Friedrich Otto, and Marcel Vollweiler

Fachbereich Elektrotechnik/Informatik, Universität Kassel
34109 Kassel, Germany
{hundeshagen,otto,vollweiler}@theory.informatik.uni-kassel.de

Abstract. We associate a *transduction* (that is, a binary relation) with the characteristic language of a restarting automaton, and we prove that in this way monotone deterministic restarting automata yield a characterization of pushdown transductions. Then we study the class of transductions that are computed by *parallel communicating systems* (PC-systems) of monotone deterministic restarting automata. We will see that this class includes all transductions that are computable.

1 Introduction

Automata with a restart operation were introduced originally to describe a method of grammar-checking for the Czech language (see, e.g., [5]). These automata started the investigation of restarting automata as a suitable tool for modeling the so-called *analysis by reduction*, which is a technique that is often used (implicitly) for developing formal descriptions of natural languages based on the notion of *dependency* [6,11]. In particular, the Functional Generative Description (FGD) for the Czech language (see, e.g., [7]) is based on this method.

FGD is a dependency based system, which translates given sentences into their underlying tectogrammatical representations, which are (at least in principle) disambiguated. Thus, the real goal of performing analysis by reduction on (the enriched form of) an input sentence is not simply to accept or reject this sentence, but to extract information from that sentence and to translate it into another form (be it in another natural language or a formal representation). Therefore, we are interested in *transductions* (that is, binary relations) and in ways to compute them by certain types of restarting automata.

Here we study two different approaches. First we associate a binary relation with the *characteristic language* of a restarting automaton, motivated by the way in which the so-called *proper language* of a restarting automaton is defined. In this way we obtain a characterization for the class of *pushdown transductions* in terms of monotone deterministic restarting automata. Then we introduce *parallel communicating systems* (PC-systems, for short) that consist of two monotone deterministic restarting automata, using them to compute transductions. In this way we obtain a characterization for the class of all computable transductions. In addition we consider the *input-output transductions* computed by these two types of restarting automata.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 163–172, 2011.

This paper is structured as follows. In Section 2 we establish basic notions concerning languages and transductions, and in Section 3 we describe in short the type of restarting automaton we use and define the associated classes of transductions. Finally, in Section 4 we introduce the announced PC-systems and establish our main results on the various classes of transductions considered. The paper closes with a short summary and some open problems.

2 Basic Notions and Definitions

Throughout the paper we will use λ to denote the empty word. Further, $|w|$ will denote the *length* of the word w, and if a is an element of the underlying alphabet, then $|w|_a$ denotes the *a-length* of w, that is, the number of occurrences of the letter a in w. Further, $\mathbb{N}_+$ will denote the set of all positive integers. By (D)CFL we denote the class of (deterministic) context-free languages.

If Σ is a subalphabet of an alphabet Γ, then by Pr^{Σ} we denote the projection from Γ^* onto Σ^*. For a language $L \subseteq \Gamma^*$, $\mathsf{Pr}^{\Sigma}(L) = \{\, \mathsf{Pr}^{\Sigma}(w) \mid w \in L \,\}$.

If Σ and Δ are two finite alphabets, then each set $R \subseteq \Sigma^* \times \Delta^*$ is called a *transduction*. For $u \in \Sigma^*$ and $v \in \Delta^*$, $R(u) = \{\, y \in \Delta^* \mid (u, y) \in R \,\}$ is the *image* of u, and $R^{-1}(v) = \{\, x \in \Sigma^* \mid (x, v) \in R \,\}$ is the *preimage* of v under R. A particular class of transductions are the *pushdown transductions*. A *pushdown transducer* (PDT for short) is defined as $T = (Q, \Sigma, \Delta, X, q_0, Z_0, F, E)$, where Q is a finite set of internal states, Σ, Δ, and X are the finite input, output, and pushdown alphabet, respectively, $q_0 \in Q$ is the initial state, $Z_0 \in X$ is the initial symbol on the pushdown store, $F \subseteq Q$ is the set of final states, and $E \subset Q \times (\Sigma \cup \{\lambda\}) \times X \times Q \times X^* \times \Delta^*$ is a finite set of transitions [1]. A *configuration* of T is written as (q, u, α, v), where $q \in Q$ is a state, $u \in \Sigma^*$ is the still unread part of the input, $\alpha \in X^*$ is the contents of the pushdown store with the first letter of α at the bottom and the last letter at the top, and $v \in \Delta^*$ is the output produced so far. If $(q, au, \alpha x, v)$ is a configuration, where $a \in \Sigma \cup \{\lambda\}$ and $x \in X$, and $(q, a, x, p, y, z) \in E$, then T can perform the transition step $(q, au, \alpha x, v) \vdash_T (p, u, \alpha y, vz)$. The transduction $\mathrm{Rel}(T)$ computed by T is defined as

$$\mathrm{Rel}(T) = \{\, (u, v) \in \Sigma^* \times \Delta^* \mid \exists q \in F,\, \alpha \in X^* :\, (q_0, u, Z_0, \lambda) \vdash_T^* (q, \lambda, \alpha, v) \,\},$$

where $\vdash_T^*$ denotes the reflexive transitive closure of the relation $\vdash_T$. A relation $R \subseteq \Sigma^* \times \Delta^*$ is a *pushdown transduction* if $R = \mathrm{Rel}(T)$ holds for some PDT T. By $\mathsf{PDR}(\Sigma, \Delta)$ we denote the class of all pushdown transductions over $\Sigma^* \times \Delta^*$.

3 Transductions Computed by Restarting Automata

A large variety of types of restarting automata has been developed over the years. Here we are only interested in the deterministic RRWW-automaton. Such an automaton consists of a finite-state control, a single flexible tape with end markers, and a read/write window of fixed size. Formally, it is described by an

8-tuple $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, where Q is a finite set of states, Σ is a finite input alphabet, Γ is a finite tape alphabet containing Σ, the symbols $¢, \$ \notin \Gamma$ are used as markers for the left and right border of the work space, respectively, $q_0 \in Q$ is the initial state, $k \geq 1$ is the size of the *read/write window*, and δ is the *transition function* that associates transition steps to pairs (q, u) consisting of a state q and a possible content u of the read/write window. There are four types of transition steps: *move-right steps* (MVR) that shift the read/write window one position to the right and change the state; *rewrite steps* that cause M to replace the contents u of its read/write window by a shorter string v, thereby reducing the length of the tape, and to change the state; *restart steps* (Restart) that cause M to place its read/write window over the left end of the tape and to reenter the initial state q_0; and *accept steps* that cause M to halt and accept. If $\delta(q, u) = \emptyset$ for some pair (q, u), then M necessarily halts, and we say that M *rejects* in this situation. We use the prefix det- to denote deterministic types of restarting automata.

A *configuration* of M is a string $\alpha q \beta$, where $q \in Q$, and either $\alpha = \lambda$ and $\beta \in \{¢\} \cdot \Gamma^* \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Gamma^*$ and $\beta \in \Gamma^* \cdot \{\$\}$; here q represents the current state, $\alpha\beta$ is the current content of the tape, and it is understood that the window contains the first k symbols of β or all of β when $|\beta| \leq k$. A *restarting configuration* is of the form $q_0 ¢ w \$$. If $w \in \Sigma^*$, then $q_0 ¢ w \$$ is an *initial configuration*.

Any computation of M consists of certain phases. A phase, called a *cycle*, starts in a restarting configuration, the head moves along the tape performing move-right steps and a single rewrite operation until a restart operation is performed and thus a new restarting configuration is reached. If no further restart operation is performed, the computation necessarily finishes in a halting configuration – such a phase is called a *tail*. It is required that in each cycle M performs exactly one rewrite step – thus each cycle strictly reduces the length of the tape. We use the notation $w \vdash_M^c z$ to denote a cycle of M that begins with the restarting configuration $q_0 ¢ w \$$ and ends with the restarting configuration $q_0 ¢ z \$$; the relation $\vdash_M^{c^*}$ is the reflexive and transitive closure of $\vdash_M^c$. Also it is required that in a tail M executes at most one rewrite step.

Let $C = \alpha q \beta$ be a *rewrite configuration* of an RRWW-automaton M, that is, a configuration in which a rewrite step is to be applied. Then the *right distance* $D_r(C)$ of C is $|\beta|$. A *sequence of rewrite configurations* $S = (C_1, C_2, \ldots, C_n)$ is called *monotone* if $D_r(C_1) \geq D_r(C_2) \geq \ldots \geq D_r(C_n)$. A *computation* of an RRWW-automaton M is called *monotone* if the sequence of rewrite configurations that is obtained from the cycles of that computation is monotone. Observe that here the rewrite configuration that corresponds to a rewrite step that is executed in the tail of a computation is not taken into account. Finally, the RRWW-automaton M is called *monotone* if each of its computations is monotone. We use the prefix mon- to denote monotone types of RRWW-automata.

A *sentential form* $w \in \Gamma^*$ *is accepted by* M, if there is an accepting computation which starts from the restarting configuration $q_0 ¢ w \$$. By $L_C(M)$ we denote the language consisting of all sentential forms accepted by M; $L_C(M)$ is the *characteristic language* of M, while the set $L(M) = L_C(M) \cap \Sigma^*$ of all

input sentences accepted by M is the *input language recognized* by M. Further, $L_{\mathrm{P}}(M) = \mathsf{Pr}^{\Sigma}(L_{\mathrm{C}}(M))$ is the *proper language* of M.

For any class A of automata, $\mathcal{L}(\mathsf{A})$ will denote the class of input languages recognizable by automata from A, and $\mathcal{L}_{\mathrm{P}}(\mathsf{A})$ will denote the class of proper languages of automata from A.

We want to study transductions that are computed by RRWW-automata. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an RRWW-automaton with tape alphabet Γ, which contains the input alphabet Σ and the *output alphabet* Δ. Here we assume that Σ and Δ are disjoint. With M we associate two transductions, where $\mathrm{sh}(u, v)$ denotes the *shuffle* of the words u and v:

$$R_{\mathrm{io}}(M) = \{\, (u, v) \in \Sigma^* \times \Delta^* \mid L_{\mathrm{C}}(M) \cap \mathrm{sh}(u, v) \neq \emptyset \,\},$$
$$R_{\mathrm{P}}(M) = \{\, (u, v) \in \Sigma^* \times \Delta^* \mid \exists w \in L_{\mathrm{C}}(M) : u = \mathsf{Pr}^{\Sigma}(w) \text{ and } v = \mathsf{Pr}^{\Delta}(w) \,\}.$$

Here $R_{\mathrm{io}}(M)$ is the *input-output transduction* of M, and $R_{\mathrm{P}}(M)$ is the *proper transduction* of M. By $\mathcal{R}el_{\mathrm{io}}(\mathsf{RRWW})$ ($\mathcal{R}el_{\mathrm{P}}(\mathsf{RRWW})$) we denote the class of input-output transductions (proper transductions) of RRWW-automata.

Theorem 1. $\mathcal{R}el_{\mathrm{P}}(\mathsf{det\text{-}mon\text{-}RRWW}) = \bigcup_{\{\Sigma, \Delta \mid \Sigma \cap \Delta = \emptyset\}} \mathsf{PDR}(\Sigma, \Delta)$.

Proof. Let Σ and Δ be disjoint alphabets. According to [10], a relation $R \subseteq \Sigma^* \times \Delta^*$ is a pushdown transduction if and only if there exists a deterministic pushdown automaton (DPDA) $P = (Q, \Gamma, X, q_0, \bot, F, \delta)$ such that $R = \{\, (\mathsf{Pr}^{\Sigma}(w), \mathsf{Pr}^{\Delta}(w)) \mid w \in L(P) \,\}$. A language $L \subseteq \Gamma^*$ is deterministic context-free if and only if there exists a monotone deterministic RRWW-automaton $M = (Q_M, \Gamma, \Gamma, ¢, \$, q_0^M, k, \delta_M)$ such that $L = L(M) = L_{\mathrm{C}}(M)$. Thus, $R \subseteq \Sigma^* \times \Delta^*$ is a pushdown transduction if and only if $R = R_{\mathrm{P}}(M)$ for some deterministic RRWW-automaton M that is monotone. □

Thus, the proper transductions of monotone deterministic RRWW-automata just describe the pushdown transductions. Next we consider the class of input-output transductions of monotone deterministic RRWW-automata.

Proposition 1. $\mathcal{R}el_{\mathrm{io}}(\mathsf{det\text{-}mon\text{-}RRWW}) \subsetneq \mathcal{R}el_{\mathrm{P}}(\mathsf{det\text{-}mon\text{-}RRWW})$.

Proof. Let $M = (Q, \Sigma, \Theta, ¢, \$, q_0^M, k, \delta)$ be a monotone deterministic RRWW-automaton, where $\Delta \subseteq \Theta \setminus \Sigma$ is an output alphabet. If we interpret $\Sigma \cup \Delta$ as input alphabet of M, then $L(M) \subseteq (\Sigma \cup \Delta)^*$ is a deterministic context-free language. Thus, there exists a monotone deterministic RRWW-automaton M' with tape alphabet $\Sigma \cup \Delta$ that accepts this language [3], that is, $L(M') = L_{\mathrm{C}}(M') = L(M)$. It follows that

$$\begin{aligned} R_{\mathrm{P}}(M') &= \{\, (\mathsf{Pr}^{\Sigma}(w), \mathsf{Pr}^{\Delta}(w)) \mid w \in L_{\mathrm{C}}(M') \,\} \\ &= \{\, (\mathsf{Pr}^{\Sigma}(w), \mathsf{Pr}^{\Delta}(w)) \mid w \in L_{\mathrm{C}}(M) \cap (\Sigma \cup \Delta)^* \,\} = R_{\mathrm{io}}(M). \end{aligned}$$

Next consider the relation $R = \{\, (a^n b^n, \lambda), (a^n b^m, \lambda) \mid n \geq 1, m > 2n \,\}$, which is easily seen to be a pushdown transduction. If M is an RRWW-automaton such that $R = R_{\mathrm{io}}(M)$, then $L(M) = \{\, a^n b^n, a^n b^m \mid n \geq 1, m > 2n \,\}$. As this

language is not accepted by any monotone deterministic RRWW-automaton [3], it follows that M is not a monotone deterministic RRWW-automaton. □

For $\Sigma \cap \Delta = \emptyset$, each *rational transduction* $R \subseteq \Sigma^* \times \Delta^*$ [2] is obviously the input-output transduction of some monotone deterministic RRWW-automaton.

4 Transformations Computed by PC-Systems of Monotone Deterministic RRWW-Automata

Instead of computing a transduction R by a single restarting automaton, we propose to compute it by a pair of restarting automata that have the ability to communicate with each other. To formalize this idea, we introduce the so-called *parallel communicating system of restarting automata* (or PC-RRWW-system, for short). A PC-RRWW-system consists of a pair $\mathcal{M} = (M_1, M_2)$ of RRWW-automata $M_i = (Q_i, \Sigma_i, \Gamma_i, ¢, \$, q_0^{(i)}, k, \delta_i)$, $i = 1, 2$. Here it is required that, for each $i \in \{1, 2\}$, the set of states Q_i of M_i contains finite subsets Q_i^{req} of *request states* of the form (q, req), Q_i^{res} of *response states* of the form $(q, \mathrm{res}(l))$, Q_i^{rec} of *receive states* of the form $(q, \mathrm{rec}(l))$, and Q_i^{ack} of *acknowledge states* of the form $(q, \mathrm{ack}(l))$. Further, in addition to the move-right, rewrite and restart steps, M_1 and M_2 have so-called *communication steps*.

A *configuration* of $\mathcal{M}$ consists of a pair $K = (k_1, k_2)$, where k_i is a configuration of M_i, $i = 1, 2$. For $w_1 \in \Gamma_1^*$ and $w_2 \in \Gamma_2^*$, the *initial configuration* on input (w_1, w_2) is $K_{\mathrm{in}}(w_1, w_2) = (q_0^{(1)}¢w_1\$, q_0^{(2)}¢w_2\$)$. The *single-step computation relation* $(k_1, k_2) \vdash_{\mathcal{M}} (k_1', k_2')$ consists of *local steps* and *communication steps*:

(Communication 1) if $k_1 = u_1(q_1, \mathrm{res}(l))v_1$ and $k_2 = u_2(q_2, \mathrm{req})v_2$, then $k_1' = u_1(q_1, \mathrm{ack}(l))v_1$ and $k_2' = u_2(q_2, \mathrm{rec}(l))v_2$;

(Communication 2) if $k_1 = u_1(q_1, \mathrm{req})v_1$ and $k_2 = u_2(q_2, \mathrm{res}(l))v_2$, then $k_1' = u_1(q_1, \mathrm{rec}(l))v_1$ and $k_2' = u_2(q_2, \mathrm{ack}(l))v_2$.

In all other cases M_1 and M_2 just perform local computation steps, independent of each other. If one of them is in a request or response state, but the other is not (yet) in the corresponding response or request state, respectively, then the latter automaton keeps on performing local steps until a communication step is enabled. Should this never happen, then the computation of $\mathcal{M}$ fails. Once one of M_1 and M_2 has accepted, the other automaton keeps on performing local steps until it either gets stuck, in which case the computation fails, or until it also accepts, in which case the computation of $\mathcal{M}$ succeeds. Hence, $R_{\mathrm{C}}(\mathcal{M}) = \{ (w_1, w_2) \in \Gamma_1^* \times \Gamma_2^* \mid K_{\mathrm{in}}(w_1, w_2) \vdash_{\mathcal{M}}^* (\mathsf{Accept}, \mathsf{Accept}) \}$ is the *characteristic transduction* of $\mathcal{M}$, $R_{\mathrm{io}}(\mathcal{M}) = R_{\mathrm{C}}(\mathcal{M}) \cap (\Sigma_1^* \times \Sigma_2^*)$ is the *input-output transduction* of $\mathcal{M}$, and $R_{\mathrm{P}}(\mathcal{M}) = \{ (\mathsf{Pr}^{\Sigma_1}(w_1), \mathsf{Pr}^{\Sigma_2}(w_2)) \mid (w_1, w_2) \in R_{\mathrm{C}}(\mathcal{M}) \}$ is the *proper transduction* of $\mathcal{M}$.

Example 1. Consider the transduction $R_{\mathrm{sort}} = \{ ((abc)^n, d^n e^n f^n) \mid n \geq 1 \}$. We describe a PC-RRWW-system $\mathcal{M} = (M_1, M_2)$ satisfying $R_{\mathrm{io}}(\mathcal{M}) = R_{\mathrm{sort}}$. Let $M_1 = (Q_1, \Sigma_1, \Sigma_1, ¢, \$, p_0, 1, \delta_1)$ and $M_2 = (Q_2, \Sigma_2, \Sigma_2, ¢, \$, q_0, 1, \delta_2)$, where

$\Sigma_1 = \{a, b, c\}$, $\Sigma_2 = \{d, e, f\}$, and the sets Q_1 and Q_2 are given implicitly by the following description of the transition functions δ_1 and δ_2:

$$
\begin{array}{lllll}
M_1: & \delta_1(p_0, ¢) & = (p_0, \mathsf{MVR}), & \delta_1(p_{b,2}, b) & = (p'_b, \mathrm{req}),\\
& \delta_1(p_0, a) & = (p_a, \mathrm{req}), & \delta_1((p'_b, \mathrm{rec}(b)), b) & = (p_{b,3}, \mathsf{MVR}),\\
& \delta_1((p_a, \mathrm{rec}(a)), a) & = (p_{a,1}, \lambda), & \delta_1(p_{b,1}, \$) & = \mathsf{Restart},\\
& \delta_1(p_{a,1}, b) & = (p_{a,2}, \mathsf{MVR}), & \delta_1(p_0, c) & = (p_c, \mathrm{req}),\\
& \delta_1(p_{a,2}, c) & = (p_{a,3}, \mathsf{MVR}), & \delta_1((p_c, \mathrm{rec}(c)), c) & = (p_{c,1}, \lambda),\\
& \delta_1(p_{a,3}, a) & = (p'_a, \mathrm{req}), & \delta_1(p_{c,1}, a) & = (p_{c,2}, \mathsf{MVR}),\\
& \delta_1((p'_a, \mathrm{rec}(a)), a) & = (p_{a,1}, \mathsf{MVR}), & \delta_1(p_{c,2}, b) & = (p_{c,3}, \mathsf{MVR}),\\
& \delta_1(p_{a,3}, \$) & = \mathsf{Restart}, & \delta_1(p_{c,3}, c) & = (p'_c, \mathrm{req}),\\
& \delta_1(p_0, b) & = (p_b, \mathrm{req}), & \delta_1((p'_c, \mathrm{rec}(c)), c) & = (p_{c,1}, \mathsf{MVR}),\\
& \delta_1((p_b, \mathrm{rec}(b)), b) & = (p_{b,3}, \lambda), & \delta_1(p_{c,1}, \$) & = (p_\$, \mathrm{req}),\\
& \delta_1(p_{b,3}, c) & = (p_{b,1}, \mathsf{MVR}), & \delta_1((p_\$, \mathrm{rec}(\$)), \$) & = \mathsf{Accept};\\
& \delta_1(p_{b,1}, a) & = (p_{b,2}, \mathsf{MVR}), & &\\
M_2: & \delta_2(q_0, ¢) & = (q_0, \mathsf{MVR}), & \delta_2(q_e, e) & = (q_e, \mathrm{res}(b)),\\
& \delta_2(q_0, d) & = (q_d, \mathrm{res}(a)), & \delta_2(q_e, f) & = (q_f, \mathrm{res}(c)),\\
& \delta_2((q_d, \mathrm{ack}(a)), d) & = (q_d, \mathsf{MVR}), & \delta_2((q_f, \mathrm{ack}(c)), f) & = (q_f, \mathsf{MVR}),\\
& \delta_2(q_d, d) & = (q_d, \mathrm{res}(a)), & \delta_2(q_f, f) & = (q_f, \mathrm{res}(c)),\\
& \delta_2(q_d, e) & = (q_e, \mathrm{res}(b)), & \delta_2(q_f, \$) & = (q_f, \mathrm{res}(\$)),\\
& \delta_2((q_e, \mathrm{ack}(b)), e) & = (q_e, \mathsf{MVR}), & \delta_2((q_f, \mathrm{ack}(\$)), \$) & = \mathsf{Accept}.
\end{array}
$$

Given $(w_1, w_2) = ((abc)^2, d^2e^2f^2)$ as input, $\mathcal{M}$ executes the computation $(p_0¢(abc)^2\$, q_0¢d^2e^2f^2\$) \vdash^*_{\mathcal{M}} (\mathsf{Accept}, \mathsf{Accept})$. Internally M_1 and M_2 check whether their inputs have the required structure, while the work of comparing the input of M_1 to the input of M_2 is completely done by communications. As M_1 and M_2 are deterministic and monotone, $R_{\mathrm{sort}} \in \mathcal{R}el_{\mathrm{io}}(\mathsf{det\text{-}mon\text{-}PC\text{-}RRWW})$.

On the other hand, R_{sort} is not the proper transduction of any monotone deterministic RRWW-automaton. Indeed, assume that $M = (Q, \Sigma_1, \Gamma, ¢, \$, q_0, k, \delta)$ is a monotone deterministic RRWW-automaton such that $R_{\mathrm{P}}(M) = R_{\mathrm{sort}}$. As M is monotone, it follows that the characteristic language $L_{\mathrm{C}}(M)$ is context-free. Now $R_{\mathrm{sort}} = R_{\mathrm{P}}(M) = \{\, (\mathsf{Pr}^{\Sigma_1}(w), \mathsf{Pr}^{\Sigma_2}(w)) \mid w \in L_{\mathrm{C}}(M) \,\}$, and hence, $\mathsf{Pr}^{\Sigma_2}(L_{\mathrm{C}}(M)) = \{\, d^n e^n f^n \mid n \geq 1 \,\}$, which is not context-free. This, however, contradicts the above observation that $L_{\mathrm{C}}(M)$ is context-free, as the class of context-free languages is closed under morphisms. Thus, we see that

$$R_{\mathrm{sort}} \in \mathcal{R}el_{\mathrm{io}}(\mathsf{det\text{-}mon\text{-}PC\text{-}RRWW}) \setminus \mathcal{R}el_{\mathrm{P}}(\mathsf{det\text{-}mon\text{-}RRWW}). \tag{1}$$

The transduction $R_{\mathrm{pal}} = \{\, (ww^R, c) \mid w \in \{a, b\}^* \,\}$ is obviously a pushdown transduction. Actually a monotone deterministic RRWW-automaton M satisfying $R_{\mathrm{io}}(M) = R_{\mathrm{pal}}$ is easily constructed. Just take the obvious RRWW-automaton with the characteristic language $L_{\mathrm{C}}(M) = \{\, wcw^R \mid w \in \{a, b\}^* \,\}$. However, the following negative result holds for this transduction.

Proposition 2. $R_{\mathrm{pal}} \notin \mathcal{R}el_{\mathrm{io}}(\mathsf{det\text{-}mon\text{-}PC\text{-}RRWW})$.

Proof. Assume that $\mathcal{M} = (M_1, M_2)$ is a PC-RRWW-system such that M_1 and M_2 are monotone and deterministic, and $R_{\mathrm{io}}(\mathcal{M}) = R_{\mathrm{pal}}$. In an accepting computation of $\mathcal{M}$, the automaton M_2 starts with tape contents $\mathbb{c}c\$$. Thus, there are only finitely many different configurations that M_2 can reach. Accordingly a *non-forgetting* RRWW-automaton M (see, e.g., [9]) can be designed that simulates $\mathcal{M}$ as follows. Here a non-forgetting RRWW-automaton is a generalization of an RRWW-automaton that is obtained by combining each restart operation with a change of the internal state just like the move-right and rewrite operations. In this way some information can be carried from one cycle to the next.

The non-forgetting RRWW-automaton M proceeds as follows. Using its tape M simulates M_1 step-by-step, while it simulates M_2 and all communication steps of $\mathcal{M}$ in its finite control. As M executes the exact cycles of M_1, it is monotone and deterministic, and it accepts the language $L(M) = \{\, ww^R \mid w \in \{a,b\}^* \,\}$. The class $\mathcal{L}$(det-mon-nf-RRWW) of languages accepted by non-forgetting monotone deterministic RRWW-automata coincides with the class of *left-to-right regular languages* [9], which is a proper subclass of the class CRL of Church-Rosser languages. Thus, $L_{\mathrm{pal}} = \{\, ww^R \mid w \in \{a,b\}^* \,\} \in \mathsf{CRL}$ follows, contradicting the fact that L_{pal} is not a Church-Rosser language [4]. □

Together with (1), this proposition shows the following.

Corollary 1. *The class of transductions* $\mathcal{R}el_{\mathrm{io}}$(det-mon-PC-RRWW) *is incomparable to the classes* $\mathcal{R}el_{\mathrm{io}}$(det-mon-RRWW) *and* $\mathcal{R}el_{\mathrm{P}}$(det-mon-RRWW) *with respect to inclusion.*

It remains to compare the classes of transductions $\mathcal{R}el_{\mathrm{io}}$(det-mon-PC-RRWW) and $\mathcal{R}el_{\mathrm{P}}$(det-mon-RRWW) to the class $\mathcal{R}el_{\mathrm{P}}$(det-mon-PC-RRWW). For doing so we will make use of the following technical result. Recall from above the notion of non-forgetting restarting automaton. A PC-RRWW-system $\mathcal{M} = (M_1, M_2)$ is called *non-forgetting* if the RRWW-automata M_1 and M_2 are non-forgetting. The following technical result shows that deterministic PC-RRWW-systems are already as expressive as deterministic non-forgetting PC-RRWW-systems in contrast to the situation for deterministic RRWW-automata (see, e.g., [8]).

Proposition 3. *For each non-forgetting deterministic* PC-RRWW*-system* $\mathcal{M}$*, there exists a deterministic* PC-RRWW*-system* $\mathcal{M}'$ *such that* $R_{\mathrm{C}}(\mathcal{M}') = R_{\mathrm{C}}(\mathcal{M})$*. In addition, if* $\mathcal{M}$ *is monotone, then so is* $\mathcal{M}'$*.*

Proof outline. Let $\mathcal{M} = (M_1, M_2)$ be a non-forgetting deterministic PC-RRWW-system. From $\mathcal{M}$ a deterministic PC-RRWW-system $\mathcal{M}' = (M_1', M_2')$ can be constructed such that M_1' and M_2' simulate M_1 and M_2, respectively, cycle by cycle. However, as M_1' and M_2' are reset to their respective initial state each time they execute a restart operation, they must determine the corresponding restart state of M_1 and M_2, respectively, by communicating with each other. In fact, whenever M_1' is about to simulate a restart step of M_1, then it determines the restart state of M_1 and sends this information to M_2'. Then, after having performed the corresponding restart step, M_1' requests the information about the correct restart state from M_2', and M_2' works similarly. There is, however,

a serious problem with this approach. At the time when M_1' sends the information about the new restart state of M_1 to M_2', the automaton M_2' may already be waiting for a communication with M_1' that simulates a communication between M_2 and M_1. Then the communication newly initiated by M_1' will not correspond to the communication expected by M_2', and consequently the system $\mathcal{M}'$ may come to a deadlock. Thus, M_1' must make sure that M_2' has not yet entered a communication before it attempts to send the information on the new restart state of M_1. Fortunately, these problems can be overcome by executing a two-way communication between M_1' and M_2' each time before a step of the computation of $\mathcal{M}$ is being simulated. This two-way communication is to ensure that both, M_1' and M_2', know the next step of both, M_1 and M_2, that they have to simulate. □

Based on Proposition 3 we obtain the following characterization.

Theorem 2. *Let $R \subseteq \Sigma^* \times \Delta^*$ be a transduction. Then R belongs to the class $\mathcal{R}el_{\mathrm{P}}(\mathsf{det\text{-}mon\text{-}PC\text{-}RRWW})$ if and only if it is computable.*

Proof. Certainly a PC-RRWW-system can be simulated by a Turing machine. Thus, given a pair $(u, v) \in R_{\mathrm{P}}(\mathcal{M})$, where $\mathcal{M}$ is a monotone deterministic PC-RRWW-system, a Turing machine T can nondeterministically guess words $x \in \Gamma_1^*$ and $y \in \Gamma_2^*$ satisfying $\mathsf{Pr}^{\Sigma}(x) = u$ and $\mathsf{Pr}^{\Delta}(y) = v$, and then it can simulate $\mathcal{M}$ starting from the initial configuration $K_{\mathrm{in}}(x, y)$. Thus, the transduction $R_{\mathrm{P}}(\mathcal{M})$ is computable.

Conversely, let $R \subseteq \Sigma^* \times \Delta^*$ be a transduction that is computable. Thus, there exists a Turing machine T_0 that, given (u, v) as input, has an accepting computation if and only if $(u, v) \in R$ holds. Actually from T_0 we obtain a nondeterministic Turing machine T_1 that, given $u \in \Sigma^*$ as input, has an accepting computation producing the result $v \in \Delta^*$ if and only if the pair (u, v) belongs to R. From T_1 we obtain a nondeterministic Turing machine $T_2 = (Q_T, \Gamma_T, q_0^{(T)}, q_+^{(T)}, \delta_T)$ by replacing the input alphabet Σ by a new alphabet $\bar{\Sigma} = \{\, \bar{a} \mid a \in \Sigma \,\}$ and by replacing the output alphabet Δ by a new alphabet $\bar{\Delta} = \{\, \bar{c} \mid c \in \Delta \,\}$ such that Σ and Δ are disjoint from Γ_T. Here we assume that $q_+^{(T)}$ is the only final state of T_2, that each accepting computation of T_2 consists of an odd number of steps, and that the tape just contains the result v (encoded as $\bar{v} \in \bar{\Delta}^*$) when T_2 halts and accepts.

From this Turing machine we now construct a monotone deterministic PC-RRWW-system $\mathcal{M} = (M_1, M_2)$ such that $R_{\mathrm{P}}(\mathcal{M}) = R$. Because of Proposition 3 we can describe $\mathcal{M}$ as a non-forgetting PC-RRWW-system. Let $\Gamma_1 = \Sigma \cup Q_T \cup \Gamma_T \cup \{\#\}$ and $\Gamma_2 = \Delta \cup Q_T \cup \Gamma_T \cup \{\#\}$ be the tape alphabets of M_1 and M_2, respectively. The characteristic transduction $R_{\mathrm{C}}(\mathcal{M})$ of $\mathcal{M}$ will consist of all pairs of words $(x, y) \in \Gamma_1^* \times \Gamma_2^*$ satisfying the following conditions:

$\exists u \in \Sigma^* \, \exists v \in \Delta^* \, \exists$ an accepting computation of T_2 of the form
$q_0^{(T)}\bar{u} \vdash_{T_2} x_1q_1y_1 \vdash_{T_2} \cdots \vdash_{T_2} x_{n-2}q_{n-2}y_{n-2} \vdash_{T_2} x_{n-1}q_{n-1}y_{n-1} \vdash_{T_2} q_+\bar{v}$:
(i) $x = \#u\#\#x_1q_1y_1\#\#x_3q_3y_3\#\# \ldots \#\#x_{n-2}q_{n-2}y_{n-2}\#\#q_+\bar{v}$, and
(ii) $y = \#\#q_0^{(T)}\bar{u}\#\#x_2q_2y_2\#\# \ldots \#\#x_{n-1}q_{n-1}y_{n-1}\#v$.

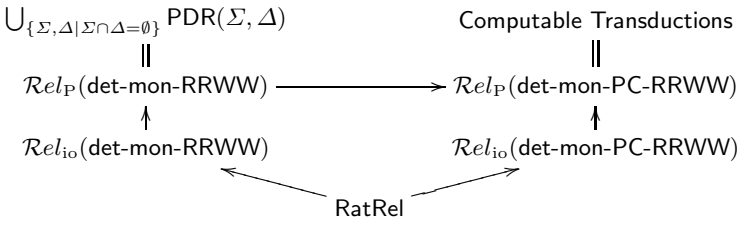


Fig. 1. Taxonomy of classes of transductions computed by various types of monotone deterministic restarting automata. Here RatRel denotes the class of rational transductions.

The non-forgetting restarting automata M_1 and M_2 behave as follows:

1. Starting from its initial state M_1 expects to have a tape contents x from the regular set $E_1 = \#\cdot\Sigma^*\cdot(\#\#\cdot\Gamma_T^*\cdot Q_T\cdot\Gamma_T^*)^*\cdot\#\#\cdot q_+^{(T)}\cdot\bar{\Delta}^*$, and M_2 expects to have a tape contents y from the regular set $E_2 = \#\#\cdot q_0^{(T)}\cdot\bar{\Sigma}^*\cdot(\#\#\cdot\Gamma_T^*\cdot Q_T\cdot\Gamma_T^*)^*\cdot\#\cdot\Delta^*$. During its first cycle M_1 erases the first occurrence of the symbol # from its tape, it checks that the factor $u \in \Sigma^*$ on its tape corresponds to the factor from $\bar{\Sigma}^*$ on M_2's tape using communications, and it verifies that its tape contents belongs to the regular language E_1. Analogously, M_2 also erases the first occurrence of the symbol # from its tape, and it verifies that its tape contents belongs to the regular set E_2. If all these tests are successful, then both M_1 and M_2 restart in particular non-initial states; otherwise, the computation fails.

2. In the next cycle M_1 and M_2 check by communication that the first factor marked by ## on M_1's tape is an immediate successor configuration of the factor marked by a single symbol # on M_2's tape with respect to the computation relation of the Turing machine T_2. During this process each of M_1 and M_2 erase the leftmost occurrence of the symbol # from their tape. In the affirmative, both M_1 and M_2 restart in non-initial states; otherwise, the computation fails.

3. In the next cycle the roles of M_1 and M_2 are interchanged.

4. Steps 2 and 3 are repeated until the syllable $q_+\bar{v}$ on M_1's tape is reached. In this case the words x and y do indeed describe an accepting computation of T_2 that produces the result $\bar{v}$ starting from $\bar{u}$. Now in a tail computation M_1 and M_2 compare the factor $\bar{v}$ on M_1's tape to the suffix $v' \in \Delta^*$ on M_2's tape by communications. If $v' = v$, then both M_1 and M_2 accept, as in this case x and y satisfy all the conditions stated above; otherwise, the computation fails.

It follows from this description that $R_{\mathrm{C}}(\mathcal{M})$ is indeed the transduction defined above, which in turn yields that $R_{\mathrm{P}}(\mathcal{M}) = R$ holds. As M_1 and M_2 are both monotone and deterministic, this completes the proof of Theorem 2. □

Together with Corollary 1 this result yields the following proper inclusions.

Corollary 2

(a) $\mathcal{R}el_{\mathrm{P}}$(det-mon-RRWW) $\subsetneq \mathcal{R}el_{\mathrm{P}}$(det-mon-PC-RRWW).
(b) $\mathcal{R}el_{\mathrm{io}}$(det-mon-PC-RRWW) $\subsetneq \mathcal{R}el_{\mathrm{P}}$(det-mon-PC-RRWW).

We summarize the relationships between the various classes of transductions considered by the diagram in Figure 1, where an arrow denotes a proper inclusion, and classes that are not connected are incomparable under inclusion.

5 Concluding Remarks

It remains to investigate the classes of input-output transductions of monotone deterministic RRWW-automata and PC-RRWW-systems in more detail. Can they be characterized by other types of transducers or through certain closure properties? The unlimited usage of auxiliary symbols to annotate the input of monotone deterministic PC-RRWW-systems in combination with the unrestricted use of communication steps gives a large computational power to these otherwise rather restricted models of automata. What class of proper transductions are computed by these very systems when we restrict the use of auxiliary symbols by a linear function in the length of the combined inputs?

References

1. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. Parsing, vol. I. Prentice-Hall, Englewood Cliffs (1972)
2. Berstel, J.: Transductions and Context-free Languages. Teubner Studienbücher, Teubner, Stuttgart (1979)
3. Jančar, P., Mráz, F., Plátek, M., Vogel, J.: On monotonic automata with a restart operation. J. Autom.Lang. Comb. 4, 283–292 (1999)
4. Jurdziński, T., Loryś, K.: Church-Rosser Languages vs. UCFL. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 147–158. Springer, Heidelberg (2002)
5. Kuboň, V., Plátek, M.: A grammar based approach to a grammar checking of free word order languages. In: COLING 1994, Proc., Kyoto, Japan, vol. II, pp. 906–910 (1994)
6. Lopatková, M., Plátek, M., Kuboň, V.: Modeling syntax of free word-order languages: Dependency analysis by reduction. In: Matoušek, V., Mautner, P., Pavelka, T. (eds.) TSD 2005. LNCS (LNAI), vol. 3658, pp. 140–147. Springer, Heidelberg (2005)
7. Lopatková, M., Plátek, M., Sgall, P.: Towards a formal model for functional generative description: Analysis by reduction and restarting automata. The Prague Bulletin of Mathematical Linguistics 87, 7–26 (2007)
8. Messerschmidt, H., Otto, F.: On deterministic CD-systems of restarting automata. Intern. J. Found. Comput. Sci. 20, 185–209 (2009)
9. Messerschmidt, H., Otto, F.: A hierarchy of monotone deterministic nonforgetting restarting automata. Theory of Coputing Systems (November 21, 2009), doi:10.1007/s00224-009-9247-x
10. Otto, F.: On proper languages and transformations of lexicalized types of automata. In: Ito, M., Kobayashi, Y., Shoji, K. (eds.) Automata, Formal Languages and Algebraic Systems, AFLAS 2008, Proc. World Scientific, Singapore (2008) (to appear)
11. Sgall, P., Hajičová, E., Panevová, J.: The Meaning of the Sentence in Its Semantic and Pragmatic Aspects. Reidel Publishing Company, Dordrecht (1986)

Uniformizing Rational Relations for Natural Language Applications Using Weighted Determinization

J. Howard Johnson

Institute for Information Technology,
National Research Council Canada,
Ottawa, Canada
Howard.Johnson@nrc-cnrc.gc.ca

Abstract. Rational functions have many applications in natural language processing. Specifying them can be difficult since many of the techniques over-generalize and incorrect transformations need to be removed or avoided. Uniformization is the process of restricting a rational relation to make it single-valued while preserving its domain. One way of doing this is to use weighted determinization with an appropriate semiring to produce a subsequential transducer when this is possible. A basic algorithm using the genealogical minimum as the selection process is discussed with a motivating example.

1 Introduction

Rational functions (single-valued finite state transductions) have many applications in the computer processing of natural language as demonstrated but the use of the Xerox finite state toolkit [2] and similar systems for morphological and phonological analysis and synthesis of many natural languages. Such toolkits have numerous techniques to help the user stay within the constraint of a functional transformation but the inherent non-determinism in many rational operators and processes leads to situations where ambiguities creep in.

A particular context where the need for such a mechanism arises is the specification of de-tokenizers for statistical machine translation (SMT). SMT works with appropriately tokenized text in a source language, translating it to tokenized text in the target language. A tokenizer is needed for the source language and a de-tokenizer for the target language.

Since it is more natural to work in terms of tokenization, de-tokenization should be expressed as the inverse operation. Furthermore, the developer of the de-tokenizer would prefer a systematic process that can be steered in an understandable manner. Choosing the shortest de-tokenized stream is often the desired output or can be easily cleaned up by a post-edit transducer. Since choosing the shortest output does not completely disambiguate the output, we will break ties by taking the lexicographic (dictionary order) minimum of the shortest output. This total order on strings is called the genealogical or radix

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 173–180, 2011.

order and the first element in a sequence ordered in such a manner is called the genealogical or radix minimum.

Johnson [6] discusses a method of disambiguating a transduction when it is being used to transform a text. This often does not provide sufficient flexibility to the programmer since it would be preferable to do the disambiguation as part of the programming process rather than once at the end.

Sakarovitch [13] discusses this process of uniformization but points out that the radix uniformization might not be achievable with a rational transducer. This is bad news for applications because it means that any algorithm will fail in the general case.

Suppose the programmer is willing to accept an algorithm that sometimes fails. Many tokenizers are inherently subsequential in execution since they do not remember more than a finite amount of left context and can manage without any lookahead. A de-tokenizer based on such a tokenizer might also be reasonably expected to be subsequential. We would like to construct a subsequential transducer that outputs the genealogical minimum of the possible outputs from a rational relation. Although it cannot be done in general, with an appropriate implementation that fails gracefully, this could be a useful tool.

Section 2 introduces some definitions and background to help make the following discussion more precise. Section 3 discusses how weighted determinization with a specially crafted semiring can be used to partially address the problem. To help clarify ideas, section 4 provides an example of the approach. Section 5 provides some concluding remarks.

2 Some Definitions and Background

Definition 1. *A* $*$-semiring S *is a set with operations* $\oplus$, $\otimes$, $^{\circledast}$, $\mathbf{0}$ *and* $\mathbf{1}$ *where:*

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z \qquad x \otimes (y \otimes z) = (x \otimes y) \otimes z$$
$$x \oplus \mathbf{0} = \mathbf{0} \oplus x = x \qquad x \otimes \mathbf{1} = \mathbf{1} \otimes x = x$$
$$x \oplus y = y \oplus x \qquad x \otimes \mathbf{0} = \mathbf{0} \otimes x = \mathbf{0}$$
$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z) \qquad (x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$$
$$(w \otimes w^{\circledast}) \oplus \mathbf{1} = (w^{\circledast} \otimes w) \oplus \mathbf{1} = w^{\circledast}$$
$$\forall x, y, z \in S, \forall w \in S - \{\mathbf{1}\}$$

Definition 2. *A* Conway $*$-semiring S *is a* $*$*-semiring with:*

$$(x \oplus y)^{\circledast} = (x^{\circledast} \otimes y)^{\circledast} \otimes x^{\circledast}$$
$$(x \otimes y)^{\circledast} = \mathbf{1} \oplus (x \otimes (y \otimes x)^{\circledast} \otimes y) \qquad \forall x, y \in S$$

when the appropriate $^{\circledast}$ *operations are defined [4].*

Definition 3. *An* idempotent $*$-semiring S *is a* $*$*-semiring with:*

$$x \oplus x = x \qquad \forall x \in S$$

Note that the set of regular languages $\mathbf{Reg}(\Sigma^*)$ over an alphabet Σ is an idempotent Conway $*$-semiring.

Definition 4. *A* totally $\preceq$-ordered alphabet Δ *is a finite set where* $\forall a, b, c \in \Delta$: $a \preceq b, b \preceq a \implies a = b$, $a \preceq b, b \preceq c \implies a \preceq c$, *and* $a \preceq b$ **or** $b \preceq a$. *We write* $a \prec b$ *if* $a \preceq b$ *and* $a \neq b$.

Definition 5. *The* genealogical order relation $\preceq$ *over* Δ^* *where* Δ *is a totally ordered alphabet is defined as:*

$$x \prec y = \begin{cases} x & \text{if } |x| < |y| \text{ or } (|x| = |y| \text{ and } x = uax_1, y = uby_1) \\ y & \text{if } |y| < |x| \text{ or } (|x| = |y| \text{ and } x = ubx_1, y = uay_1) \end{cases}$$

where $x, y, u, x_1, y_1 \in \Delta^*$, $a, b \in \Delta$, *and* $a \prec b$. $x \preceq y$ *if* $x \prec y$ *or* $x = y$.

Definition 6. $\mathbf{GM}(\Delta^*)$ *for an alphabet* Δ *is the set* $\Delta^* \cup \perp$ *with the operations* $\oplus$, $\otimes$, $^{\circledast}$, $\mathbf{0}$, $\mathbf{1}$*:*

$$x \oplus y = \begin{cases} x & \text{if } x \preceq y \text{ or } y = \perp \\ y & \text{if } y \preceq x \text{ or } x = \perp \end{cases} \qquad x \otimes y = \begin{cases} xy & \text{if } x, y \in \Delta^* \\ \perp & \text{if } x = \perp \text{ or } y = \perp \end{cases}$$

$$x^{\circledast} = \epsilon \qquad \mathbf{0} = \perp \qquad \mathbf{1} = \epsilon$$

$\mathbf{GM}(\Delta^*)$ can easily be shown to be an idempotent Conway $*$-semiring since it is the homomorphic image of $\mathbf{Reg}(\Delta^*)$ where the $\emptyset$ maps to $\perp$ and other languages are mapped to their genealogical minimal element [6].

Definition 7. *The GCLD* $x \wedge y = z$ *of two elements of a* $*$*-semiring is a left divisor that is maximal. That is* $x = z \otimes x_1$, $y = z \otimes y_1$, *and if there is a* v *such that* $x = v \otimes x_2$, $y = v \otimes y_2$ *then* $z = v \otimes z_1$. *Here* x_1, y_1, x_2, y_2, *and* z_1 *are all from the* $*$*-semiring.*

We will refer to a $*$-semiring where $\wedge$ is defined for any two pair of elements as a GCLD $*$-semiring.

Definition 8. *A* weighted finite automaton A *over alphabet* Σ *and weight space* S *is a 7-tuple:*

$$A = \langle Q, \Sigma, S, I, F, E, \lambda, \rho \rangle$$

where Q *is a finite set of states,* Σ *is a finite alphabet,* S *is a* $*$*-semiring,* $I \subseteq Q$ *is set of initial states,* $F \subseteq Q$ *is set of final states,* $E \subseteq Q \times (\Sigma \cup \epsilon) \times S \times Q$ *is a set of transitions,* λ *is a function from initial states to* S, *and* ρ *is a function from final states to* S.

Definition 9. *A* finite state transducer T *with input alphabet* Σ *and output alphabet* Δ *is*

$$T = \langle Q, \Sigma, \mathbf{Reg}(\Delta^*), I, F, E, \lambda, \rho \rangle$$

where Q *is a finite set of states,* Σ *and* Δ *are finite alphabets, and I, F, E,* λ, *and* ρ *are as above with* $S = \mathbf{Reg}(\Delta^*)$.

Although standard definitions of finite state transducers choose output transitions from Δ^* without loss of expressive power, this characterization is equivalent to the usual ones [3] and opens the door to weighted finite automata with weights chosen from $\mathbf{GM}(\Delta^*)$. Furthermore, any transducer can be converted to such a weighted automaton by applying the necessary homomorphic mapping to the weights.

3 Algorithm

The $\mathbf{GM}(\Delta^*)$ $*$-semiring is combined with the implementation of weighted determinization as discussed by Mohri [12] to yield a method that uniformizes rational relations to (not necessarily finite) subsequential transducers. Note that there are cases where the result of the process is subsequential but a finite transducer is not found without further modification to the algorithm.

For now we will live with the fact that weighted determinization is a semi-algorithm that either computes a subsequential transducer or runs forever exploring more and more of the infinite machine until computing resources are exhausted. We can live with the nice terminating cases that cover many practical situations, or follow Mohri by treating weighted determinization as a lazy algorithm that only expands states that are used by a particular application that only visits a finite part of the infinite state-space.

First of all our rational relation must be expressed in the form of a transducer (Definition 10). Each weight is replaced by its natural homomorphic image in $\mathbf{GM}(\Delta^*)$.

Before applying the weighted determinization algorithm, the automaton must be cleaned up a bit. It must be *trim*, *ϵ-free*, and *accelerated* through the application of weight pushing.

For a weighted automaton to be *trim*, any states that are not reachable by a path from a start state are removed together with any transitions they carry. Next, any states from which a final state cannot be reached are removed together with any transitions they carry. Requiring accessibility of states is not important for us except to reduce the sizes of data structures; however, the presence of non co-accessible states can drastically affect the result leading to infinite machines where the algorithm would otherwise converge to a finite solution.

To be *ϵ-free*, we must first recognize that any transition with only an output is an ϵ-transition and its 'weight' must be aggregated into preceding non-ϵ-transitions using semiring operations. We are helped here by the fact that $\mathbf{GM}(\Delta^*)$ is a k-closed semiring with $k = 0$. Effectively, this means that any ϵ-loops evaluate to a weight of ϵ, the multiplicative identity and can be simply deleted. There are no further problems with ϵ-removal. The easiest general algorithms work in an obvious way.

To *accelerate* the automaton through weight pushing there are some more fundamental changes that need to be made. We will paraphrase Mohri with appropriate modifications:

> Let A be a weighted automaton over a semiring S. Assume that S is a $GCLD$ $*$-semiring. *(This can be weakened further if necessary.)* For any state $q \in Q$, assume that the following sum is defined and in S:
>
> $$d[q] = \bigoplus_{\pi \in P(q,F)} (w[\pi] \otimes \rho(n[\pi])).$$
>
> $d[q]$ is the *weighted distance* from q to F including the final weight and is well defined for all $q \in Q$ when S is a k-closed semiring. The weight

pushing algorithm consists of computing each weighted distance $d[q]$ and of *re-weighting* the transition weights, initial weights, and final weights in the following way:

$$\begin{aligned} \forall e \in E \text{ s.t. } d[p[e]] = \mathbf{0}, \quad & w[e] \leftarrow d[p[e]] \backslash (w[e] \otimes d[n[e]], \\ \forall q \in I, \quad & \lambda(q) \leftarrow \lambda(q) \otimes d[q], \\ \forall q \in F, \text{ s.t. } d[q] \neq \mathbf{0}, \quad & \rho(q) \leftarrow d[q] \backslash \rho(q). \end{aligned}$$

Here $p[e]$, $n[e]$, $w[e]$ are the source, destination, weight respectively of e.

We are now ready to do weighted determinization using a suitably modified version of Mohri's presented as Algorithm 1. Note that the same changes made above are again necessary. Line 11 requires that the Greatest Common Left Divisor of the considered weights must be calculated. In Mohri's case, he can choose divisors more freely and chooses a sum. Here we must ensure left divisibility and choose the maximal element that still left divides. The change in line 12 involves replacing a left multiplication of an inverse by a straightforward left division. Left division is defined in this case where an inverse doesn't exist.

There are also a couple of less important differences in the calculation of I' and λ' in lines 3 to 6. This is a small nicety that factors out as much output as can be emitted before any input is read. This bizarre idea of writing output before reading anything usually doesn't occur in practical applications but results in a small reduction in the size of the resulting machine.

Note that, following Mohri, the notation $Q[p']$ means the states in p', $E[Q[p']]$ are the transitions have a tail in a state of p', $i[E[Q[p']]]$ are the labels form Σ in transitions have have a tail in a state of p', $i[e]$ is the label form Σ from transition e, and $n[e]$ is the destination state from transition e.

There remains one more step in the usual determinization suite. There often is a benefit in minimizing the resulting machine by combining states that have equivalent right context. The generalization of the conventional minimization algorithm for unweighted finite state machines works correctly if the pair of letter and weight from each transition is treated as an element of an expanded alphabet $\Sigma \times S$. Mohri says that weight pushing should be performed before minimization. In the situation described here this will be unnecessary because we applied weight pushing before determinization and the algorithm preserves the effect.

4 An Example

Suppose that we have text that contains four types of information: (1) Words made up of upper and lower case letters. We will restrict our alphabet to 'a' and its upper case form 'A'. (2) Numbers are a sequence of digits starting with a non-zero digit. We will restrict our digits to '0' and '1'. (3) Punctuation are individual punctuation marks. Each punctuation mark will be a separate token. We will restrict ourselves to ',' and '.'. (4) White space is a sequence of blanks.

Algorithm 1. WEIGHTED-DETERMINIZATION(A)

$A \equiv \langle Q, \Sigma, S, I, F, E, \lambda, \rho \rangle$
$Q' \leftarrow \emptyset$, $F' \leftarrow \emptyset$, $E' \leftarrow \emptyset$, $\lambda' \leftarrow \emptyset$, $\rho' \leftarrow \emptyset$, $\mathcal{Z} \leftarrow$ empty QUEUE
$w' \leftarrow \bigwedge\{\lambda(q) : q \in I\}$
$q' \leftarrow \{(q, w' \backslash \lambda(q)) : q \in I\}$
$I' \leftarrow \{q'\}$
$\lambda'(q') \leftarrow w'$
$\mathcal{Z} \leftarrow$ ENQUEUE($\mathcal{Z}, q'$)
while NOTEMPTY($\mathcal{Z}$) **do**
    $p' \leftarrow$ DEQUEUE($\mathcal{Z}$)
    for each $x \in i[E[Q[p']]]$ **do**
        $w' \leftarrow \bigwedge\{v \otimes w : (p, v) \in p', (p, x, w, q) \in E\}$
        $q' \leftarrow \{(q, \bigoplus\{w' \backslash (v \otimes w) : (p, v) \in p', (p, x, w, q) \in E\})$
            $: q = n[e], i[e] = x, e \in E[Q[p']]\}$
        $E' \leftarrow E' \cup \{(p', x, w', q')\}$
        if $q' \notin Q'$ **then**
            $Q' \leftarrow Q' \cup \{q'\}$
            if $Q[q'] \cap F \neq \emptyset$ **then**
                $F' \leftarrow F' \cup \{q'\}$
                $\rho'(q') \leftarrow \bigoplus\{v \otimes \rho(q) : (q, v) \in q', q \in F\}$
            ENQUEUE($\mathcal{Z}, q'$)
return $A' \equiv \langle Q', \Sigma, S, I', F', E', \lambda', \rho' \rangle$

We wish to construct a subsequential transducer that tokenizes the text in the following way: (1) Words are converted to lower case. (2) Numbers are copied. (3) Punctuation marks are copied. (4) White space is converted to a single blank. Every token in the tokenized text is separated by exactly one blank, whether there is white space occurring in the input or not. Extra blanks are inserted appropriately. Word and number tokens must be maximal. We also will produce a de-tokenizer from our specification that produces the genealogical minimum as output. An INR specification for the required transducer follows:

```
Upper = { A };          Lower = { a };          PosDigit = { 1 };
Digit = { 0, 1 };       Punct = { ',', '.' };   Blank = ' ';
Token = ( Upper | Lower )+;    Number = PosDigit Digit*;
White = Blank+;                ToLower = { ( A, a ), ( a, a ) }*;
TCopy = ( Token @@ ToLower )  [[ T ]];
NCopy = ( Number $ ( 0, 0 ) ) [[ N ]];
PCopy = ( Punct $ ( 0, 0 ) )  [[ P ]];
WCopy = ( White, Blank )      [[ W ]];
Binst = ( , Blank )           [[ B ]];
Copy = [[ S ]] ( TCopy | NCopy | PCopy | WCopy | Binst )* [[ E ]];
ZAlph = { T, N, P, W, B, S, E };
Invalid = { T, N, P } { T, N, P };
PreTokenize = Copy @ ( ZAlph* Invalid ZAlph* :acomp );
Tokenize    = PreTokenize :GMsseq;
DeTokenize  = Tokenize $ ( 1, 0 ) :GMsseq;
```

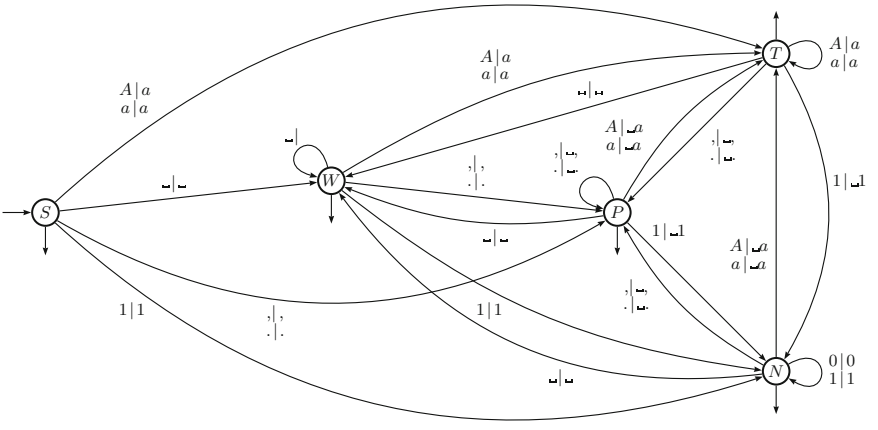


Fig. 1. Tokenizer

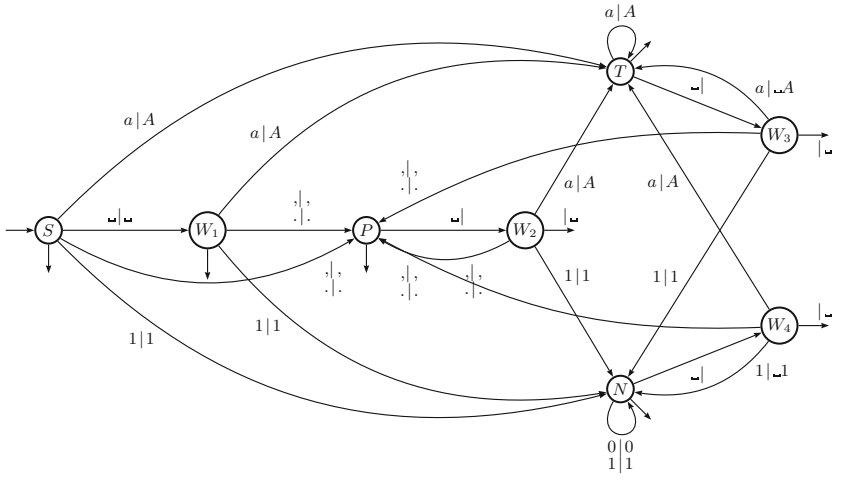


Fig. 2. De-tokenizer

Some of the techniques used by INR are unusual. A three-tape transducer with the third tape used for control is used here. Letters T, N, P, W, B, C, E (in double brackets to select tape 3) are tokens that appear on the control tape. The composition operator (denoted by '@') causes the control tape to be removed after the constraint of disallowing 'Invalid' sequences. Figure 1 shows the result of the application of Algorithm 1 and Figure 2 shows a de-tokenizer that results from applying Algorithm 1 to the inverse relation from Figure 1.

Finally, here is an example where the result of weighted determinization is subsequential but Algorithm 1 does not terminate. Suppose that in the above example, we insist that a word can be either all upper-case or all lower case and mixed case is disallowed. The de-tokenizer, faced with a word token of arbitrary length in lower case, would have to decide whether to output the upper case form or the lower case form. Of course the answer would be upper case since

the ASCII letter ordering is being used. However, the decision about output is deferred until the token is complete, and with arbitrarily long input, will be deferred forever.

5 Conclusion and Future Work

A useful tool for uniformizing finite state transductions can be implemented using a variation of weighted determinization over a novel $*$-semiring.

Practically and theoretically speaking it is unsatisfying to have a procedure that fails by running until resources are exhausted. It would be definitely superior to terminate if the computation is not possible, provide some diagnostic information, and give a result that is still usable though with some flaws. In addition the case where the expected result is subsequential but the algorithm fails should be addressed.

References

1. Abdali, S.K., Saunders, B.D.: Transitive closure and related semiring properties via eliminants. Theoretical Computer Science 40, 257–274 (1985)
2. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI Publications, Stanford (2003), http://www.fsmbook.com
3. Berstel, J.: Transductions and context-free languages. Teubner, Stuttgart (1979)
4. Conway, J.H.: Regular algebra and finite machines. Chapman and Hall, London (1971)
5. Johnson, J.H.: INR—a program for computing finite state automata. INR manual (1988), http://ginr.org
6. Johnson, J.H.: A unified framework for disambiguating finite transductions. Theoretical Computer Science 63, 91–111 (1989)
7. Lehmann, D.J.: Algebraic structures for transitive closure. Theoretical Computer Science 4(1), 59–76 (1977)
8. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23(2), 269–312 (1997)
9. Mohri, M.: Generic ϵ-removal algorithm for weighted automata. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 230–242. Springer, Heidelberg (2001)
10. Mohri, M., Pereira, F., Riley, M.: The design principles of a weighted finite-state transducer library. Theoretical Computer Science 231(1), 17–32 (2000)
11. Mohri, M.: Generic ϵ-removal and input ϵ-normalization algorithms for weighted transducers. International Journal of Foundations of Computer Science (2002)
12. Mohri, M.: Weighted automata algorithms. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata, ch. 6, pp. 213–254. Springer, Berlin (2009)
13. Sakarovitch, J.: Elements of automata theory. Cambridge University Press, Cambridge (2009)

Partially Ordered Two-Way Büchi Automata*

Manfred Kufleitner and Alexander Lauser

FMI, Universität Stuttgart, Germany
{kufleitner,lauser}@fmi.uni-stuttgart.de

Abstract. We introduce partially ordered two-way Büchi automata over infinite words. As for finite words, the nondeterministic variant recognizes the fragment Σ_2 of first-order logic FO[<] and the deterministic version yields the Δ_2-definable ω-languages. As a byproduct of our results, we show that deterministic partially ordered two-way Büchi automata are effectively closed under Boolean operations.

In addition, we have coNP-completeness results for the emptiness problem and the inclusion problem over deterministic partially ordered two-way Büchi automata.

1 Introduction

Büchi automata have been introduced in order to decide monadic second-order logic over infinite words [2]. Today, they have become one of the most important tools in model-checking sequential finite state systems, see e.g. [1,3]. Büchi automata are nondeterministic finite automata, accepting infinite words if there exists an infinite run such that some final state occurs infinitely often. A generalization are two-way Büchi automata; Pécuchet showed that they have the same expressive power as ordinary Büchi automata [10]. Alternating two-way Büchi automata have been used for model checking of temporal logic formulas with past modalities [7,16]. These automata, too, can recognize nothing but regular ω-languages. With the usual padding technique, the succinctness result for two-way automata over finite words [5] immediately yields an exponential lower bound for the succinctness of two-way Büchi automata.

We introduce partially ordered two-way (po2) Büchi automata and we characterize their expressive power in terms of fragments of first-order logic FO[<]. The fragment Σ_2 consists of all FO[<]-sentences in prenex normal form with one block of existential quantifiers followed by one block of universal quantifiers followed by a propositional formula. The fragment Π_2 contains the negations of Σ_2-formulas. By abuse of notation, we identify logical fragments with the classes of ω-languages they define. Hence, it makes sense to define $\Delta_2 = \Sigma_2 \cap \Pi_2$, i.e., an ω-language is Δ_2-definable if it is both Σ_2-definable and Π_2-definable. Therefore, Δ_2 is the largest subclass of Σ_2 (or Π_2) which is closed under complementation. Various characterizations of Σ_2 and of Δ_2 over infinite words are

* This work was supported by the German Research Foundation (DFG), grant DI 435/5-1.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 181–190, 2011.

known [15,4]. Requiring that a Σ_2-formula and a Π_2-formula agree on all infinite words is in some sense more restrictive than requiring that they agree on all finite words. For example over finite words, Δ_2 has the same expressive power as first-order logic with only two variables [14], whereas over infinite words, Δ_2 is weaker than first-order logic with two variables [4]. Moreover, Δ_2 over finite words coincides with a language class called *unambiguous polynomials* [11], whereas over infinite words, only some restricted variant of unambiguous polynomials is definable in Δ_2 [4].

Schwentick, Thérien, and Vollmer introduced the po2-automaton model over finite words [12]; cf. [8] for further characterizations of such automata. A po2-automaton is a two-way automaton with the property that once a state is left, it is never entered again. Every such automaton admits a partial order on its states such that transitions are non-decreasing. In fact, one could use a linear order on the states, but this would distort the length of a longest chain, which in some cases is a useful parameter. Nondeterministic po2-automata recognize exactly the Σ_2-definable languages over finite words whereas deterministic po2-automata correspond to Δ_2-definable languages [12].

In this paper, we present analog results over infinite words. More precisely, for $L \subseteq \Gamma^\omega$ we show that

- L is recognized by some nondeterministic partially ordered two-way Büchi automaton if and only if L is definable in Σ_2 (Theorem 1),
- L is recognized by some deterministic partially ordered two-way Büchi automaton if and only if L is definable in Δ_2 (Theorem 3).

In particular, nondeterministic po2-Büchi automata are more powerful than deterministic po2-Büchi automata, and nondeterministic po2-Büchi automata are not closed under complementation. The proof of Theorem 1 is a straightforward generalization of the respective result for finite words. It is presented here for the sake of completeness. The proof of Theorem 3 is new. It is based on a language description from [4] rather than on so called *turtle languages* as in [12]. The main step in our proof is to show that deterministic po2-Büchi automata are effectively closed under Boolean operations (Theorem 2). This is non-trivial, since the approach of starting a second automaton after the first one has completed its computation does not work for Büchi automata. To this end, we simulate two deterministic po2-Büchi automata simultaneously, and we have to do some bookkeeping of positions if the two automata walk in different directions. Based on a *small model property* of po2-Büchi automata, we show in Theorem 4 that various decision problems over po2-Büchi automata are coNP-complete: the emptiness problem for deterministic and for nondeterministic po2-Büchi automata; and the universality, the inclusion, and the equivalence problem for deterministic po2-Büchi automata. Note that for (non-partially-ordered) one-way Büchi automata, both the inclusion problem and the equivalence problem are PSPACE-complete [13].

Due to lack of space, some proofs are omitted. For complete proofs, we refer to the full version of this paper [6].

2 Preliminaries

Throughout this paper, Γ denotes a finite alphabet. The set of finite words over $A \subseteq \Gamma$ is A^* and the set of infinite words over A is A^ω. If we want to emphasize that $\alpha \in \Gamma^\omega$ is an infinite word, then we say that α is an *ω-word*. The empty word is ε. We have $\emptyset^* = \{\varepsilon\}$ and $\emptyset^\omega = \emptyset$. The *length* of a finite word $w \in \Gamma^*$ is denoted by $|w|$, i.e., $|w| = n$ if $w = a_1 \cdots a_n$ with $a_i \in \Gamma$. We set $|\alpha| = \infty$ if $\alpha \in \Gamma^\omega$. The *alphabet* of a word $\alpha = a_1 a_2 \cdots \in \Gamma^* \cup \Gamma^\omega$ is denoted by $\mathrm{alph}(\alpha)$. It is the set of letters occurring in α. We say that a position i of α is an *a-position* of α if $a_i = a$.

A *language* is a subset of Γ^* or a subset of Γ^ω. We emphasize that $L \subseteq \Gamma^\omega$ contains only infinite words by saying that L is an *ω-language*. A *monomial* (of *degree* k) is a language of the form $P = A_1^* a_1 \cdots A_k^* a_k A_{k+1}^*$. It is *unambiguous* if each word $w \in P$ has a unique factorization $w = u_1 a_1 \cdots u_k a_k u_{k+1}$ with $u_i \in A_i^*$. Similarly, an *ω-monomial* is an ω-language of the form $Q = A_1^* a_1 \cdots A_k^* a_k A_{k+1}^\omega$ and it is *unambiguous* if each word $\alpha \in Q$ has a unique factorization $u_1 a_1 \cdots u_k a_k \beta$ with $u_i \in A_i^*$ and $\beta \in A_{k+1}^\omega$. A *restricted unambiguous ω-monomial* is an unambiguous ω-monomial $A_1^* a_1 \cdots A_k^* a_k A_{k+1}^\omega$ such that $\{a_i, \ldots, a_k\} \not\subseteq A_i$ for all $1 \leq i \leq k$. A *polynomial* is a finite union of monomials and an *ω-polynomial* is a finite union of ω-monomials. A *restricted unambiguous ω-polynomial* is a finite union of restricted unambiguous ω-monomials.

By FO[$<$] we denote the first-order logic over words interpreted as labeled linear orders. As atomic formulas, FO[$<$] comprises $\top$ (for *true*) and $\bot$ (for *false*), the unary predicate $\lambda(x) = a$ for $a \in \Gamma$, and the binary predicate $x < y$ for variables x and y. The idea is that variables range over the linearly ordered positions of a word and $\lambda(x) = a$ means that x is an a-position. Apart from the Boolean connectives, we allow quantifications over position variables, i.e., existential quantifications $\exists x\colon \varphi$ and universal quantifications $\forall x\colon \varphi$ for $\varphi \in$ FO[$<$]. The semantics is as usual.

Every formula in FO[$<$] can be converted into a semantically equivalent formula in prenex normal form by renaming variables and moving quantifiers to the front. This gives rise to the fragment Σ_2 (resp. Π_2) consisting of all FO[$<$]-formulas in prenex normal form with only two blocks of quantifiers, starting with a block of existential quantifiers (resp. universal quantifiers). Note that the negation of a formula in Σ_2 is equivalent to a formula in Π_2 and vice versa. The fragments Σ_2 and Π_2 are both closed under conjunction and disjunction.

A *sentence* in FO[$<$] is a formula without free variables. Since there are no free variables in a sentence φ, the truth value of $\alpha \models \varphi$ is well-defined. The *ω-language defined by* φ is $L(\varphi) = \{\alpha \in \Gamma^\omega \mid \alpha \models \varphi\}$. We frequently identify logical fragments with the respective classes of languages. For example, $\Delta_2 = \Sigma_2 \cap \Pi_2$ consist of all languages L such that $L = L(\varphi) = L(\psi)$ for some $\varphi \in \Sigma_2$ and $\psi \in \Pi_2$, i.e., a language L is Δ_2-definable if there are equivalent formulas in Σ_2 and in Π_2 defining L. The notion of *equivalence* depends on the models and it turns out to be a difference whether we use finite or infinite words as models, cf. [4,14]. Unless stated otherwise, we shall only use infinite word models. In particular, for the remainder of this paper Δ_2 is a class of ω-languages.

2.1 Partially Ordered Two-Way Büchi Automata

In the following, we give the Büchi automaton pendant of a two-way automaton. This is basically a Büchi automaton that may change the direction in which it reads the input. A *two-way Büchi automaton* $\mathcal{A} = (Z, \Gamma, \delta, X_0, F)$ is given by:

- a finite set of states $Z = X \dot\cup Y$,
- a finite input alphabet Γ; the tape alphabet is $\Gamma \dot\cup \{\triangleright\}$, where the left end marker $\triangleright$ is a new symbol,
- a transition relation $\delta \subseteq (Z \times \Gamma \times Z) \cup (Y \times \{\triangleright\} \times X)$,
- a set of initial states $X_0 \subseteq X$, and
- a set of final states $F \subseteq Z$.

The states Z are partitioned into "ne$X$t-states" X and "Yesterday-states" Y. The idea is that states in X are entered with a right-move of the head while states in Y are entered with a left-move. For $(z, a, z') \in \delta$ we frequently use the notation $z \xrightarrow{a} z'$. On input $\alpha = a_1 a_2 \cdots \in \Gamma^\omega$ the tape is labeled by $\triangleright\alpha$, i.e., positions $i \geq 1$ are labeled by a_i and position 0 is labeled by $\triangleright$. A *configuration* of the automaton is given by a pair (z, i) where $z \in Z$ is a state and $i \in \mathbb{N}$ is the current position of the head. A *transition* $(z, i) \vdash (z', j)$ on configurations (z, i) and (z', j) exists, if

- $z \xrightarrow{a} z'$ for some $a \in \Gamma \cup \{\triangleright\}$ such that i is an a-position, and
- $j = i + 1$ if $z' \in X$, and $j = i - 1$ if $z' \in Y$.

The $\triangleright$-position can only be encountered in a state from Y and left via a state from X. In particular, $\mathcal{A}$ can never overrun the left end marker $\triangleright$. Due to the partition of the states Z, we can never have a change in direction without changing the state. A configuration (z, i) is *initial*, if $z \in X_0$ and $i = 1$. A *computation* of $\mathcal{A}$ on input α is an infinite sequence of transitions

$$(z_0, i_0) \vdash (z_1, i_1) \vdash (z_2, i_2) \vdash \cdots$$

such that (z_0, i_0) is initial. It is *accepting*, if there exists some final state which occurs infinitely often in this computation. Now, $\mathcal{A}$ *accepts* an input α if there is an accepting computation of $\mathcal{A}$ on input α. As usual, the language recognized by $\mathcal{A}$ is $L(\mathcal{A}) = \{\alpha \in \Gamma^\omega \mid \mathcal{A} \text{ accepts } \alpha\}$.

A two-way Büchi automaton is *deterministic* if $|X_0| = 1$ and if for every state $z \in Z$ and every symbol $a \in \Gamma \cup \{\triangleright\}$ there is at most one $z' \in Z$ with $z \xrightarrow{a} z'$. A two-way Büchi automaton is *complete* if for every state $z \in Z$ and every symbol $a \in \Gamma$ there is at least one $z' \in Z$ with $z \xrightarrow{a} z'$, and for every $z \in Y$ there is at least one $z' \in X$ with $z \xrightarrow{\triangleright} z'$.

We are now ready to define *partially ordered* two-way Büchi automata. We use the abbreviation "po2" for "partially ordered two-way". A two-way Büchi automaton $\mathcal{A}$ is a *po2-Büchi automaton*, if there is a partial order $\preccurlyeq$ on the set of states Z such that every transition is non-descending, i.e., if $z \xrightarrow{a} z'$ then $z \preccurlyeq z'$. In po2-Büchi automata, every computation enters a state at most once and it defines a non-decreasing sequence of states. Since there can be no

infinite chain of states, every computation has a unique state $z \in Z$ which occurs infinitely often and this state is maximal among all states in the computation. Moreover, $z \in X$ since the automaton cannot loop in a left-moving state forever. We call this state z *stationary*. A computation is accepting if and only if its stationary state z is a final state. In particular, we can always assume $F \subseteq X$ in po2-Büchi automata.

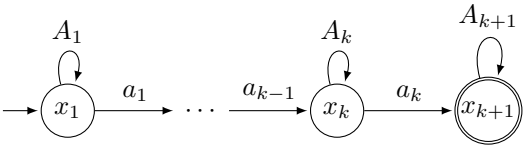
3 Nondeterministic po2-Büchi Automata

In this section, we show that nondeterministic po2-Büchi automata recognize exactly the class of Σ_2-definable languages. Moreover, it turns out that nondeterministic po2-Büchi automata and nondeterministic partially ordered one-way Büchi automata (i.e., $Y = \emptyset$ in our definition of nondeterministic po2-Büchi automata) have the same expressive power. The proof is a straightforward extension of the respective result for finite words [12]. It is presented here only for the sake of completeness.

Theorem 1. *Let $L \subseteq \Gamma^\omega$. The following assertions are equivalent:*

1. *L is recognized by a nondeterministic po2-Büchi automaton.*
2. *L is Σ_2-definable.*
3. *L is recognized by a nondeterministic partially ordered Büchi automaton.*

Proof. "1 ⇒ 2": Let $\mathcal{A}$ be a partially ordered two-way Büchi automaton. It suffices to show that $L(\mathcal{A})$ is an ω-polynomial, since every ω-polynomial is Σ_2-definable. This follows from Lemma 1 below (with $\mathcal{A} = \mathcal{B}$). "2 ⇒ 3": Every Σ_2-definable ω-language is an ω-polynomial [15]. The following Büchi automaton recognizes the ω-monomial $A_1^* a_1 \cdots A_k^* a_k A_{k+1}^\omega$:



Now, every ω-polynomial can be recognized by a finite union of such automata. "3 ⇒ 1": Every partially ordered one-way Büchi automaton is a special case of a po2-Büchi automaton. □

Lemma 1. *Let $\mathcal{A}$ and $\mathcal{B}$ be complete po2-Büchi automata and let $n_\mathcal{A}$ and $n_\mathcal{B}$ be the lengths of the longest chains in the state sets of $\mathcal{A}$ and $\mathcal{B}$, respectively. Then for every $\alpha \in L(\mathcal{A}) \cap L(\mathcal{B})$ there exists an ω-monomial P_α of degree at most $n_\mathcal{A} + n_\mathcal{B} - 2$ such that $\alpha \in P_\alpha \subseteq L(\mathcal{A}) \cap L(\mathcal{B})$. In particular,*

$$L(\mathcal{A}) \cap L(\mathcal{B}) \quad = \bigcup_{\alpha \in L(\mathcal{A}) \cap L(\mathcal{B})} P_\alpha$$

is an ω-polynomial, since there are only finitely many ω-monomials of degree at most $n_\mathcal{A} + n_\mathcal{B} - 2$.

4 Deterministic po2-Büchi Automata

This section contains the main contribution of our paper, namely that the class of languages recognizable by deterministic po2-Büchi automata is exactly the fragment Δ_2 of first-order logic. Our proof relies on a characterization of Δ_2 in terms of restricted unambiguous ω-polynomials [4]. As an intermediate step, we show in Theorem 2 that deterministic po2-Büchi automata are effectively closed under Boolean operations. Closure under complementation is surprising in the sense that for general deterministic one-way Büchi automata (not necessarily partially ordered), the same result does not hold.

Theorem 2. *The class of languages recognized by deterministic po2-Büchi automata is effectively closed under complementation, union, and intersection.*

Proof. For the effective closure under complementation we observe that the unique stationary state determines the acceptance of the input word. Therefore, complementation is achieved by complementing the set of final states. Effective closure under positive Boolean combinations is Proposition 1. □

Proposition 1. *The class of languages recognized by deterministic po2-Büchi automata is effectively closed under union and intersection.*

Proof. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be complete deterministic po2-Büchi automata. We give a product automaton construction $\mathcal{A}$ recognizing $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. With a different choice of the final states, the same automaton also recognizes $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$. We start with a description of the general idea of our construction. Details are given below. The automaton $\mathcal{A}$ operates in two modes: the *synchronous mode* and the *asynchronous mode*. In the synchronous mode $\mathcal{A}$ executes both automata at the same time until at least one of them changes to a left-moving state. Then $\mathcal{A}$ changes to the asynchronous mode by activating a left-moving automaton and suspending the other one. The position where this divergence happens is called the *synchronization point*. We stay in the asynchronous mode until the synchronization point is reached again. In a complete partially ordered automaton this must happen eventually. If the two automata now agree on going to the right, we switch back to the synchronous mode; else the process is repeated.

In order to recognize the synchronization point while executing the active automaton in the asynchronous mode, $\mathcal{A}$ administers a stack of letters and a pointer on this stack. The stack records the letters which led to a state change during synchronous mode in at least one of the automata. The corresponding positions of the word are called *marker positions* and its labels are *markers*. Let $a_1 \cdots a_m$ be the sequence of markers encountered during the computation and let $p_1 < \cdots < p_m$ be the respective marker positions. Changing from synchronous mode to asynchronous mode involves a state change of one of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$. In particular, if $\mathcal{A}$ is in the asynchronous mode, then a_m is the label of the synchronization point p_m. Since both automata are deterministic, we have that for every $1 \leq k \leq m$ the prefix of the input of length p_k is the shortest prefix admitting $a_1 \cdots a_k$ as a (scattered) subword. Our construction takes advantage

of this observation for detecting the synchronization point and in order to keep the pointer up to date while simulating the active automaton. The semantics of the pointer is as follows: If it points to a marker a_k in an X-state (i.e., the current state was entered with a right-move of the head) then the current position q of $\mathcal{A}$ is in the left-open interval $(p_{k-1}; p_m]$ and $a_k \cdots a_m$ is a scattered subword of the factor induced by the interval $[q; p_m]$. If it points to a_k in a Y-state then $q \in [p_{k-1}; p_m)$ and $a_k \cdots a_m$ is a scattered subword of $(q; p_m]$. Here, we set $p_0 = 0$ to be the position of the left end marker $\triangleright$ for convenience. If the automaton is in an X-state, scans a_m and the pointer points to the top of the stack, then we can deduce $q = p_m$, i.e., that we have reached the synchronization point. Now, if $\mathcal{A}$ is in a Y-state at an a_{k-1}-position and moves to the left afterward, then it is quite possible that we are to the left of p_{k-1}. But we cannot be to the left of p_{k-2} and we know that now the subword $a_{k-1} \cdots a_m$ appears in $[q; p_m]$. Thus we adjust the pointer to a_{k-1} in this case. On the other hand, if we scan a_k in an X-state, then we know that we are at a position $\geq p_k$ since a_k cannot appear in the interval $(p_{k-1}; p_k)$. Moreover, the subword $a_{k+1} \cdots a_m$ still appears in $(q; p_m]$. Therefore, we adjust the pointer to a_{k+1}, if after reading a_k the automaton moves to the right.

What follows are the technical details of this construction. For $i \in \{1, 2\}$ let $\mathcal{A}_i = (Z_i, \Gamma, \delta_i, x_i^0, F_i)$ with $Z_i = X_i \dot\cup Y_i$. We construct $\mathcal{A} = (Z, \Gamma, \delta, x^0, F)$ with $Z = X \dot\cup Y$ satisfying the following constraints:

- $Z \subseteq (\Gamma^* \times X_1 \times X_2) \cup (\Gamma^* \times Z_1 \times Z_2 \times \mathbb{N} \times \{\mathcal{A}_1, \mathcal{A}_2\})$. The states of the first term in the union are for the synchronous mode. The first component is the stack of markers. Its size is bounded by $|X_1| + |X_2|$. For the asynchronous states, the fourth component is the pointer to the stack of markers and the fifth component specifies the active automaton.
- $Y = Z \cap \big((\Gamma^* \times Y_1 \times Z_2 \times \mathbb{N} \times \{\mathcal{A}_1\}) \cup (\Gamma^* \times Z_1 \times Y_2 \times \mathbb{N} \times \{\mathcal{A}_2\})\big)$ and $X = Z \setminus Y$. So the left-moving states of $\mathcal{A}$ are exactly those where in asynchronous mode the active component is left-moving.
- $x^0 = (\varepsilon, x_1^0, x_2^0)$, i.e., at the beginning $\mathcal{A}$ is in the synchronous mode, the stack of markers is empty, and both automata are in their initial state.
- For recognizing the intersection we set $F = Z \cap (\Gamma^* \times F_1 \times F_2)$. For recognizing the union we set $F = Z \cap \big((\Gamma^* \times F_1 \times X_2) \cup (\Gamma^* \times X_1 \times F_2)\big)$.

Next, we describe the transitions $z \xrightarrow{a} z'$ of $\mathcal{A}$. Let $z = (w, z_1, z_2)$ when $\mathcal{A}$ is in synchronous mode, and $z = (w, z_1, z_2, k, \mathcal{C})$ otherwise. Furthermore, let $z_1 \xrightarrow{a} z_1'$ in $\mathcal{A}_1$ and let $z_2 \xrightarrow{a} z_2'$ in $\mathcal{A}_2$. Suppose that $\mathcal{A}$ is in synchronous mode, i.e., $z \in \Gamma^* \times X_1 \times X_2$. Let $w' = w$ if $z_1' = z_1$ and $z_2' = z_2$, and $w' = wa$ otherwise, i.e., push the symbol to the stack if the state of at least one automaton changes its state. We set

$$(w, z_1, z_2) \xrightarrow{a} \begin{cases} (w', z_1', z_2') & \text{if } z_1' \in X_1 \text{ and } z_2' \in X_2, \\ (w', z_1', z_2, |w'|, \mathcal{A}_1) & \text{if } z_1' \in Y_1, \\ (w', z_1, z_2', |w'|, \mathcal{A}_2) & \text{else,} \end{cases}$$

i.e., we stay in synchronous mode if both automata agree on moving right for the next step, we suspend the second automaton if $\mathcal{A}_1$ wants to move to the left

(independent of the direction of $\mathcal{A}_2$), and we suspend the first automaton when it wants to move to the right but $\mathcal{A}_2$ wants to move to the left. Consider now an asynchronous state $z \in \Gamma^* \times Z_1 \times Z_2 \times \mathbb{N} \times \{\mathcal{A}_1, \mathcal{A}_2\}$. First we deal with the special case of which may lead to a synchronization. Let $z \in X$ be scanning the top letter of the stack, i.e., a is the last letter of w and the pointer is $|w|$:

$$(w, z_1, z_2, |w|, \mathcal{C}) \xrightarrow{a} \begin{cases} (w, z_1', z_2') & \text{if } z_1' \in X_1 \text{ and } z_2' \in X_2, \\ (w, z_1', z_2, |w|, \mathcal{A}_1) & \text{if } z_1' \in Y_1, \\ (w, z_1, z_2', |w|, \mathcal{A}_2) & \text{else.} \end{cases}$$

The first case is that both automata now agree on the direction of moving to the right and then we change to synchronous mode. If not, the right-moving automaton is suspended. If both are left-moving, then $\mathcal{A}_2$ is suspended. For the other situations we only consider the case of $\mathcal{C} = \mathcal{A}_1$ being active. The case $\mathcal{C} = \mathcal{A}_2$ is similar.

$$(w, z_1, z_2, k, \mathcal{A}_1) \xrightarrow{a} \begin{cases} (w, z_1', z_2, k-1, \mathcal{A}_1) & \text{if } z_1, z_1' \in Y_1 \text{ and } a_{k-1} = a, \\ (w, z_1', z_2, k+1, \mathcal{A}_1) & \text{if } z_1, z_1' \in X_1 \text{ and } a_k = a, \\ (w, z_1', z_2, k \quad\;, \mathcal{A}_1) & \text{else.} \end{cases}$$

Since $\mathcal{A}_1$ is active, we simulate this automaton. The fourth component never gets greater than $|w|$, since scanning the last remaining symbol in an X-state is treated differently.

One can verify that $\mathcal{A}$ is partially ordered. The main idea is that between any increase and any decrease of the pointer (and also between any decrease and any increase), the state of the active automaton changes.

Let n_1 and n_2 be the length of a maximal chain of states in X_1 and X_2, respectively. The size of the stack in the first component is bounded by $n = n_1 + n_2 - 2$. Therefore, the construction can be realized by an automaton with at most $|\Gamma|^n |Z_1||Z_2|(1 + 2n)$ states. Moreover, the construction is effective. □

Proposition 2. *Every restricted unambiguous ω-monomial is recognized by a deterministic po2-Büchi automaton.*

Lemma 2. *Let $\mathcal{A}$ be a deterministic po2-Büchi automaton. Then $L(\mathcal{A})$ is a restricted unambiguous ω-polynomial.*

Theorem 3. *Let $L \subseteq \Gamma^\omega$. The following assertions are equivalent:*

1. *L is recognized by a deterministic po2-Büchi automaton.*
2. *L is Δ_2-definable.*

Proof. An ω-language L is Δ_2-definable if and only if L is a restricted unambiguous ω-polynomial [4]. The implication "1 ⇒ 2" is Lemma 2, and "2 ⇒ 1" follows from Proposition 1 and Proposition 2. □

Example 1. The ω-language $\{a, b\}^* a \emptyset^* c \{c\}^\omega$ is recognizable by a deterministic po2-Büchi automaton, but it is not recognizable by a deterministic partially ordered *one-way* Büchi automaton. Hence, the class of ω-languages recognizable by deterministic partially ordered one-way Büchi automata is a strict subclass of the class recognizable by deterministic po2-Büchi automata. ◊

5 Complexity Results

In this section, we prove some complexity bounds for the following decision problems (given po2-Büchi automata $\mathcal{A}$ and $\mathcal{B}$):

- Inclusion: Decide whether $L(\mathcal{A}) \subseteq L(\mathcal{B})$.
- Equivalence: Decide whether $L(\mathcal{A}) = L(\mathcal{B})$.
- Emptiness: Decide whether $L(\mathcal{A}) = \emptyset$.
- Universality: Decide whether $L(\mathcal{A}) = \Gamma^\omega$.

Lemma 3. Inclusion *is in* coNP *for nondeterministic* $\mathcal{A}$ *and deterministic* $\mathcal{B}$.

Lemma 4. Emptiness *is* coNP*-hard for deterministic po2-Büchi automata.*

Theorem 4. Emptiness *is* coNP*-complete for both nondeterministic and deterministic po2-Büchi automata.* Inclusion*,* Equivalence *and* Universality *are* coNP*-complete for deterministic po2-Büchi automata; for* Inclusion *this still holds for nondeterministic* $\mathcal{A}$*.*

Proof. Taking $L(\mathcal{B}) = \emptyset$, Lemma 3 yields that Emptiness is in coNP for nondeterministic po2-Büchi automata. Lemma 4 shows that Emptiness is coNP-hard even for deterministic po2-Büchi automata.

From Inclusion $\in$ coNP for deterministic po2-Büchi automata, we immediately get that Equivalence and Universality are in coNP. Moreover, the trivial reductions from Emptiness to Universality to Equivalence and from Emptiness to Inclusion show that all problems under consideration are coNP-hard for deterministic po2-Büchi automata.

For nondeterministic $\mathcal{A}$ and deterministic $\mathcal{B}$, Lemma 3 shows that Inclusion is in coNP and of course it is coNP-hard since this is already true if both automata are deterministic. □

6 Conclusion

In this paper, we introduced partially ordered two-way Büchi automata (po2-Büchi automata). The nondeterministic variant corresponds to the fragment Σ_2 of first-order logic, whereas the deterministic variant is characterized by the fragment $\Delta_2 = \Sigma_2 \cap \Pi_2$. The characterization of nondeterministic automata uses similar techniques as for finite words [12]. For deterministic automata, our proof uses new techniques and it relies on a novel language description of Δ_2 involving restricted unambiguous ω-polynomials [4]. As an intermediate step it turns out that the class of ω-languages recognized by deterministic po2-Büchi automata is effectively closed under Boolean operations.

The complexity of the Emptiness problem for both deterministic and nondeterministic po2-Büchi automata is coNP-complete. For deterministic po2-Büchi automata the decision problems Inclusion, Equivalence, and Universality are coNP-complete. To date, no non-trivial upper bounds are known for these decision problems over nondeterministic automata. Moreover, the complexity of the decision problems for general two-way Büchi automata as well as the succinctness of this model have not yet been considered in the literature.

Considering fragments with successor would be a natural extension of our results. An automaton model for the fragment Δ_2 with successor over finite words has been given by Lodaya, Pandya, and Shah [9] in terms of *deterministic partially ordered two-way automata with look-around*. We conjecture that extending such automata with a Büchi acceptance condition yields a characterization of Δ_2 with successor over infinite words.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
2. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Proc. Int. Congr. for Logic, Methodology, and Philosophy of Science, pp. 1–11. Stanford Univ. Press, Stanford (1962)
3. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
4. Diekert, V., Kufleitner, M.: Fragments of first-order logic over infinite words. In: STACS 2009. Dagstuhl Seminar Proceedings, vol. 09001, pp. 325–336 (2009)
5. Kapoutsis, C.A.: Removing bidirectionality from nondeterministic finite automata. In: Jedrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 544–555. Springer, Heidelberg (2005)
6. Kufleitner, M., Lauser, A.: Partially ordered two-way Büchi automata. Technical report no. 2010/03, Universität Stuttgart, Informatik (2010)
7. Kupferman, O., Piterman, N., Vardi, M.Y.: Extended temporal logic revisited. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 519–535. Springer, Heidelberg (2001)
8. Lodaya, K., Pandya, P.K., Shah, S.S.: Marking the chops: An unambiguous temporal logic. IFIP TCS 273, 461–476 (2008)
9. Lodaya, K., Pandya, P.K., Shah, S.S.: Around dot depth two. In: Gao, Y., Lu, H., Seki, S., Yu, S. (eds.) DLT 2010. LNCS, vol. 6224, pp. 305–316. Springer, Heidelberg (2010)
10. Pécuchet, J.-P.: Automates boustrophédon et mots infinis. Theoretical Computer Science 35, 115–122 (1985)
11. Pin, J.-É., Weil, P.: Polynomial closure and unambiguous product. Theory of Computing Systems 30(4), 383–422 (1997)
12. Schwentick, T., Thérien, D., Vollmer, H.: Partially-ordered two-way automata: A new characterization of DA. In: Kuich, W., Rozenberg, G., Salomaa, A. (eds.) DLT 2001. LNCS, vol. 2295, pp. 239–250. Springer, Heidelberg (2002)
13. Sistla, A.P., Vardi, M.Y., Wolper, P.L.: The complementation problem for Büchi automata with applications to temporal logic. Theoretical Computer Science 49(2-3), 217–237 (1987)
14. Thérien, D., Wilke, T.: Over words, two variables are as powerful as one quantifier alternation. In: STOC 1998, pp. 234–240 (1998)
15. Thomas, W.: Classifying regular events in symbolic logic. Journal of Computer and System Sciences 25, 360–376 (1982)
16. Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998)

Two-Party Watson-Crick Computations

Martin Kutrib and Andreas Malcher

Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
{kutrib,malcher}@informatik.uni-giessen.de

Abstract. We investigate synchronous systems consisting of two finite automata running in opposite directions on a shared read-only input. The automata communicate by sending messages. The communication is quantitatively measured by the number of messages sent during a computation. It is shown that even the weakest non-trivial devices in question, that is, systems that are allowed to communicate constantly often only, accept non-context-free languages. We investigate the computational capacity of the devices in question and prove a strict four-level hierarchy depending on the number of messages sent. The strictness of the hierarchy is shown by means of Kolmogorov complexity. For systems with unlimited communication several properties are known to be undecidable. A question is to what extent communication has to be reduced in order to regain decidability. Here, we derive that the problems remain non-semidecidable even if the communication is reduced to a limit close to the logarithm of the length of the input. Furthermore, we show that the border between decidability and undecidability is crossed when the communication is reduced to be constant. In this case only semilinear languages can be accepted.

1 Introduction

Watson-Crick automata were introduced in [3] as a formal model for DNA computing. Their definition has been inspired by processes observed in nature and laboratories. The idea is to have an automaton with two reading heads running on either strand of a double stranded DNA-molecule. Since in nature enzymes that actually move along DNA strands may obey the biochemical direction of the single strands of the DNA sequence, so-called $5' \to 3'$ Watson-Crick automata have been introduced in [12] after an idea presented in [13]. Basically, these systems are two-head finite automata where the heads start at opposite ends of a strand and move in opposite physical directions. If the complementarity relation of the double stranded sequence is known to be one-to-one, no additional information is encoded in the second strand. Then $5' \to 3'$ Watson-Crick automata share a common input sequence.

Whenever several heads of a device are controlled by a common finite-state control, one may suppose that the heads are synchronous and autonomous finite automata that communicate their states in every time step. Here, we add a new feature to $5' \to 3'$ Watson-Crick automata. It seems to be unlikely that in reality

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 191–200, 2011.

enzymes moving along and acting on DNA molecules communicate in every time step. So, we consider $5' \to 3'$ Watson-Crick systems where the components may but don't need to communicate by broadcasting messages. We are interested in the impact of communication in such devices, where the communication is quantitatively measured by the total number of messages sent during a computation. The role of message complexity in conventional one-way and two-way multi-head finite automata has been studied in [6,7], where deep results have been obtained. According to the notations given there and in order to differentiate the notation from 'conventional' $5' \to 3'$ Watson-Crick automata we call the devices in question two-party Watson-Crick systems. Recently, deterministic Watson-Crick automata have been studied from a descriptional complexity point of view in [2].

The idea of another related approach is based on so-called parallel communicating finite automata systems which were introduced in [10]. In this model, the input is read and processed in parallel by several one-way (left-to-right) finite automata. The communication is defined in such a way that an automaton can request the current state from another automaton, and is set to that state after receiving it whereby its former state is lost. One can distinguish whether each automaton which sends its current state is reset to its initial state or not. The degree of communication in such devices was studied in [11]. Without considering the message complexity, the concept of one-way parallel communicating finite automata systems was investigated for conventional Watson-Crick automata in [1].

In this paper, we study the computational capacity and decidability questions for deterministic two-party Watson-Crick systems whose message complexity is bounded. It is shown that devices allowed to send just one message during a computation can accept non-regular languages. For at most two messages we obtain a non-context-free language. More general, we prove a strict four-level hierarchy depending on the number of messages sent, where the levels are given by $O(1)$, $O(\log(n))$, $O(\sqrt{n})$, and $O(n)$ messages allowed. The strictness of the hierarchy is shown by arguments of Kolmogorov complexity. For systems with unlimited communication several properties are known to be undecidable [8]. Here, we prove that problems such as emptiness, finiteness, infiniteness, inclusion, equivalence, regularity, and context-freeness are non-semidecidable, that is, not recursively enumerable even if the communication is reduced to a limit $O(\log(n) \cdot \log\log(n))$. However, we show that problems become decidable when the communication is reduced to be constant. In this case only semilinear languages can be accepted. The proof is based on a simulation of deterministic two-party Watson-Crick systems by deterministic reversal-bounded multi-counter machines.

2 Preliminaries and Definitions

We denote the set of nonnegative integers by $\mathbb{N}$. We write Σ^* for the set of all words over the finite alphabet Σ. The empty word is denoted by λ, and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. The reversal of a word w is denoted by w^R and for the length of w we write $|w|$. We use $\subseteq$ for inclusions and $\subset$ for strict inclusions.

A two-party Watson-Crick system is a device of two finite automata working independently and in opposite directions on a common read-only input data. The automata communicate by broadcasting messages. The transition function of a single automaton depends on its current state, the currently scanned input symbol, and the message currently received from the other automaton. Both automata work synchronously and the messages are delivered instantly. Whenever the transition function of (at least) one of the single automata is undefined the whole systems halts. The input is accepted if at least one of the automata is in an accepting state. A formal definition is as follows.

Definition 1. *A* deterministic two-party Watson-Crick system (*DPWK*) *is a construct* $\mathcal{A} = \langle \Sigma, M, \rhd, \lhd, A_1, A_2, \rangle$, *where* Σ *is the finite set of* input symbols, M *is the set of possible* messages, $\rhd \notin \Sigma$ *and* $\lhd \notin \Sigma$ *are the* left and right endmarkers, *and each* $A_i = \langle Q_i, \Sigma, \delta_i, \mu_i, q_{0,i}, F_i \rangle$, $i \in \{1, 2\}$, *is basically a* deterministic finite automaton *with* state set Q_i, initial state $q_{0,i} \in Q_i$, *and set of* accepting states $F_i \subseteq Q_i$. *Additionally, each* A_i *has a* broadcast function $\mu_i : Q_i \times (\Sigma \cup \{\rhd, \lhd\}) \to M \cup \{\perp\}$ *which determines the message* to be sent, *where* $\perp \notin M$ *means* nothing to send, *and a* (partial) transition function $\delta_i : Q_i \times (\Sigma \cup \{\rhd, \lhd\}) \times (M \cup \{\perp\}) \to Q_i \times \{0, 1\}$, *where* 1 *means to move the head one square and* 0 *means to keep the head on the current square.*

The automata A_1 and A_2 are called *components* of the system $\mathcal{A}$, where the so-called *upper* component A_1 starts at the left end of the input and moves from left to right, and the *lower* component A_2 starts at the right end of the input and moves from right to left. A *configuration* of $\mathcal{A}$ is represented by a string $\rhd v_1 \overrightarrow{p} x v_2 y \underleftarrow{q} v_3 \lhd$, where $v_1 x v_2 y v_3$ is the input and it is understood that component A_1 is in state p with its head scanning symbol x, and component A_2 is in state q with its head scanning symbol y. System $\mathcal{A}$ starts with component A_1 in its initial state scanning the left endmarker and component A_2 in its initial state scanning the right endmarker. So, for input $w \in \Sigma^*$, the initial configuration is $\overrightarrow{q_{0,1}} \rhd w \lhd \underleftarrow{q_{0,2}}$. A computation of $\mathcal{A}$ is a sequence of configurations beginning with an initial configuration. One step from a configuration to its successor configuration is denoted by $\vdash$. Let $w = a_1 a_2 \cdots a_n$ be the input, $a_0 = \rhd$, and $a_{n+1} = \lhd$, then we set $a_0 \cdots a_{i-1} \overrightarrow{p} a_i \cdots a_j \underleftarrow{q} a_{j+1} \cdots a_{n+1} \vdash a_0 \cdots a_{i'-1} \overrightarrow{p_1} a_{i'} \cdots a_{j'} \underleftarrow{q_1} a_{j'+1} \cdots a_{n+1}$, for $0 \leq i \leq j \leq n+1$, and $a_0 \cdots a_j \underleftarrow{q} a_{j+1} \cdots a_{i-1} \overrightarrow{p} a_i \cdots a_{n+1} \vdash a_0 \cdots a_{j'} \underleftarrow{q} a_{j'+1} \cdots a_{i'-1} \overrightarrow{p} a_{i'} \cdots a_{n+1}$, for $0 \leq j \leq i \leq n+1$, iff $\delta_1(p, a_i, \mu(q, a_j)) = (p_1, d_1)$ and $\delta_2(q, a_j, \mu(p, a_i)) = (q_1, d_2)$, $i' = i + d_1$ and $j' = j - d_2$. As usual we define the reflexive, transitive closure of $\vdash$ by $\vdash^*$.

A computation *halts* when the successor configuration is not defined for the current configuration. This may happen when the transition function of one component is not defined. The language $L(\mathcal{A})$ accepted by a DPWK $\mathcal{A}$ is the set of inputs $w \in \Sigma^*$ such that there is some computation beginning with the initial configuration for w and halting with at least one component being in an accepting state.

In the following, we study the impact of communication in deterministic two-party Watson-Crick systems. The communication is measured by the total number of messages sent during a computation, where it is understood that $\perp$ means no message and, thus, is not counted.

Let $f : \mathbb{N} \to \mathbb{N}$ be a mapping. If all $w \in L(\mathcal{A})$ are accepted with computations where the total number of messages sent is bounded by $f(|w|)$, then $\mathcal{A}$ is said to be *communication bounded by* f. We denote the class of DPWKs that are communication bounded by f by DPWK(f).

In general, the *family of languages accepted* by devices of type X is denoted by $\mathscr{L}(X)$. To illustrate the definitions we start with a result giving an example.

Lemma 2. *The language* $\{ a^n b^n c^n \mid n \geq 1 \}$ *belongs to* $\mathscr{L}(DPWK(2))$.

Proof. The principal idea of the construction is that both components move their heads with different speeds from left to right and from right to left, respectively. The lower component sends a message when the borders between b's and c's and between a's and b's have been crossed, whereas the upper component checks whether it receives both messages at the correct time. More detailed, on input of the form $a^i b^j c^k$ the upper component starts to move across the a-block, while the lower component moves across the c-block. When the lower component reaches a symbol b for the first time it sends a message and continues to move to the left with half speed, that is, one square at every other time step. When it arrives at a symbol a for the first time it sends a second message. After receiving the first message exactly at the first symbol b, the upper component continues to move to the right one square in each time step. It accepts the input if and only if it receives the second message exactly when it reaches the right endmarker. In this case we know $i = k$ by the first message. The second message ensures $2j = j + k$ and, thus, $j = k$. □

3 Computational Capacity

First we note that any DPWK can be simulated by a two-way two-head finite automaton in a straightforward manner. Therefore, the family $\mathscr{L}$(DPWK) is a proper subclass of the complexity class L. From Lemma 2 we can immediately derive the construction of a DPWK(1) that accepts the non-regular language $\{ a^n b^n \mid n \geq 1 \}$. Together with the obvious inclusion of the regular languages in $\mathscr{L}$(DPWK(1)) we obtain that just one communication suffices to accept all regular languages and, additionally, also some non-regular languages, whereas two communications allow to accept non-context-free languages. However, the witness languages are semilinear. In [8] it has been shown that DPWKs that communicate in every time step accept non-semilinear languages. So, the question arises how much communication is necessary to accept a non-semilinear language. The next lemma gives an upper bound. A lower bound will be derived later.

Lemma 3. *Language* $L_{expo} = \{ a^{2^0} b a^{2^2} b \cdots b a^{2^{2m}} c a^{2^{2m+1}} b \cdots b a^{2^3} b a^{2^1} \mid m \geq 1 \}$ *belongs to* $\mathscr{L}(DPWK(O(\log(n))))$.

Proof. Roughly, the idea of the construction is that in a first phase the components compare the lengths 2^0 with 2^1, 2^2 with 2^3, ..., and 2^{2m} with 2^{2m+1}. After the first phase both components have reached the center symbol c. Next, a second phase is used to compare the length 2^{2m} with 2^{2m-1}, 2^{2m-2} with 2^{2m-3},..., and 2^2 with 2^1. To this end, the lower component A_2 waits on the c until the upper component A_1 has moved across the block $a^{2^{2m+1}}$. For the comparisons, A_1 moves across its a-blocks with half speed, while A_2 moves across its a-blocks one square in each step. The lengths of the first and second phase are checked by communicating when a b or c is reached which must happen synchronously.

The length of an accepted input is $n = 2^{2m+2} + 2m$. There is a communication on every symbol b and on symbol c as well as on the right endmarker, so there are $2m + 2$ communications. This is of order $O(\log(n))$. □

By a similar construction as for Lemma 3 the next lemma can be shown.

Lemma 4. *The language* $L_{poly} = \{\, aba^5ba^9b \cdots ba^{4m+1}ca^{4m+3}b \cdots ba^{11}ba^7ba^3 \mid m \geq 0 \,\}$ *belongs to* $\mathscr{L}(DPWK(O(\sqrt{n})))$.

As another important example we consider the language $\{\, wcw^R \mid w \in \{0,1\}^* \,\}$. As already mentioned in [8], it is accepted by a DPWK with unlimited communication, that is, by a DPWK($O(n)$).

Lemma 5. *The language* $\{\, wcw^R \mid w \in \{0,1\}^* \,\}$ *belongs to* $\mathscr{L}(DPWK(O(n)))$.

By definition we have a finite hierarchy of language classes as follows.

$$\mathscr{L}(\mathrm{DPWK}(O(1))) \subset \mathscr{L}(\mathrm{DPWK}(O(\log(n)))) \subset \mathscr{L}(\mathrm{DPWK}(O(\sqrt{n}))) \subset \mathscr{L}(\mathrm{DPWK}(O(n)))$$

Next, we turn to separate the levels of the hierarchy, that is, we show that the inclusions are, in fact, strict. For the proof of the following theorem we use an incompressibility argument. General information on Kolmogorov complexity and the incompressibility method may be found in [9]. Let $w \in \{0,1\}^+$ be an arbitrary binary string of length n. Then the Kolmogorov complexity $C(w|n)$ of w denotes the minimal size of a program describing w and knowing the length n. It is well known that there exist binary strings w of arbitrary length n such that $C(w|n) \geq n$ (cf. [9]).

Theorem 6. *If* $f \in \frac{n}{\omega(\log(n))}$, *then* $L = \{\, wcw^R \mid w \in \{0,1\}^* \,\}$ *does not belong to* $\mathscr{L}(DPWK(f))$.

Proof. By way of contradiction, we assume that L is accepted by some DPWK(f) $\mathcal{A} = \langle \Sigma, M, \rhd, \lhd, A_1, A_2, \rangle$ with $f(n) \in \frac{n}{\omega(\log(n))}$. Let $z = wcw^R$ for some $w \in \{0,1\}^+$ and $K_0 \vdash \cdots \vdash K_{acc}$ be the accepting computation on input z where K_0 is the initial configuration and K_{acc} is an accepting configuration.

Next, we make snapshots of configurations at every time step a communication takes place. For every such configuration, we remember the time step t_i, the

current states $q_1^{(i)}$, $q_2^{(i)}$ and head positions $p_1^{(i)}$, $p_2^{(i)}$ of the components, and both messages sent $(m_1^{(i)}, m_2^{(i)})$. Thus, the ith snapshot is denoted by the tuple $(t_i, q_1^{(i)}, p_1^{(i)}, m_1^{(i)}, q_2^{(i)}, p_2^{(i)}, m_2^{(i)})$. Since there are altogether at most $f(2|w|+1)$ communications, the list of snapshots Λ contains at most $f(2|w|+1)$ entries.

We claim that each snapshot can be represented by at most $O(\log(|w|))$ bits. It can be shown that acceptance is in linear time and, therefore, each time step can be represented by at most $O(\log(|w|))$ bits. Each position can also be represented by at most $O(\log(|w|))$ bits. Finally, each state and message obtained can be represented by a constant number of bits. Altogether, each snapshot can be represented by $O(\log(|w|))$ bits. So, the list Λ can be represented by at most $f(2|w|+1) \cdot O(\log(|w|)) = \frac{|w|}{\omega(\log(|w|))} \cdot O(\log(|w|)) = o(|w|)$ bits.

Next, we claim that the list Λ of snapshots together with snapshots of K_0 and K_{acc} and the knowledge of $\mathcal{A}$ and $|w|$ is sufficient to reconstruct w. This reconstruction is described by the following program P which may be realized, e.g., by some Turing machine. First, P sequentially simulates $\mathcal{A}$ on all $2^{|w|}$ inputs xcx^R where $|x| = |w|$. Additionally, it is checked whether the computation simulated has the same snapshots of the initial configuration, all communication configurations, and the accepting configuration. In this way, the string w can be identified. We have to show that there is no other string $w' \neq w$ which can be identified in this way as well. Let us assume that such a w' exists. Then all snapshots of accepting computations on input wcw^R and $w'cw'^R$ are identical. This means that both computations start and end at the same time steps and both components are in the same state and position. Additionally, in both computations communications take place at the same time steps, both components are in the same state and position at that moment and obtain the same messages. Then both computations are also accepting on input wcw'^R which is a contradiction.

Thus, w can be reconstructed given the above program P, the list of snapshots Λ, snapshots of the initial and accepting configuration, $\mathcal{A}$, and $|w|$. Since the size of P and $\mathcal{A}$ is bounded by a constant, the size of Λ is bounded by $o(|w|)$, and $|w|$ as well as the size of both remaining snapshots is bounded by $O(\log(|w|))$ each, we can reconstruct w from a description of total size $o(|w|)$. Hence, the Kolmogorov complexity $C(w||w|)$, that is, the minimal size of a program describing w is bounded by the size of the above description, and we obtain $C(w||w|) \in o(|w|)$. On the other hand, we know that there are binary strings w of arbitrary length such that $C(w||w|) \geq |w|$. This is a contradiction for w being long enough. □

The previous theorem separates the language classes $\mathscr{L}(\mathrm{DPWK}(O(\sqrt{n})))$ and $\mathscr{L}(\mathrm{DPWK}(O(n)))$ by the witness language $\{ wcw^R \mid w \in \{0,1\}^* \}$. The next theorem separates the classes $\mathscr{L}(\mathrm{DPWK}(O(\log(n))))$ and $\mathscr{L}(\mathrm{DPWK}(O(\sqrt{n})))$ by the witness language $\hat{L}_{poly} = \{ ax_1a^5x_2 \cdots x_m a^{4m+1} c a^{4m+3} x_m \cdots x_2 a^7 x_1 a^3 \mid m \geq 0 \text{ and } x_i \in \{0,1\}, 1 \leq i \leq m \}$ which can be shown to belong to the language class $\mathscr{L}(\mathrm{DPWK}(O(\sqrt{n})))$. This proof and the proof the next theorem is omitted owing to space constraints. The remaining two levels are separated at the end of Section 4.

Theorem 7. *If $f \in O(\log(n))$, then $\hat{L}_{poly}$ does not belong to $\mathscr{L}(DPWK(f))$.*

4 Decidability Problems

For DPWKs with unlimited communication, that is for DPWK($O(n)$)s, emptiness, finiteness, equivalence, and inclusion are shown to be undecidable in [8]. This section is devoted to investigating decidability problems of DPWKs with sparse communication. The question is to what extent communication has to be reduced in order to regain the decidability of certain problems. Here, we derive that the problems remain undecidable even if the communication is reduced to a limit close to the logarithm of the length of the input. Furthermore we show that the border between decidability and undecidability is crossed when the communication is reduced to be constant.

To prove our non-semidecidability results we use the technique of valid computations of Turing machines [4]. Here, a decidability problem is said to be *semidecidable* if the set of all instances for which the answer is "yes" is recursively enumerable. It suffices to consider deterministic Turing machines with one single read-write head and one single tape whose space is fixed by the length of the input, that is, so-called linear bounded automata (LBA).

Basically, it has been shown in [8] that, given a Turing machine M the set of valid computations VALC(M) is accepted by some DPWK with unlimited communication, that is, by some DPWK($O(n)$). We omit the details of the definition of the slightly modified set VALC$_{ex}$ which can be shown to be accepted by some DPWK($\log(n) \cdot \log\log(n)$).

Theorem 8. *The problems of testing emptiness, finiteness, and infiniteness are not semidecidable for DPWK*$(\log(n) \cdot \log\log(n))$.

Proof. We prove the theorem by reduction of the decidability problems for LBAs. To this end, let M be an arbitrary LBA as it has been used for the definition of the valid computations. First, M is modified to accept not before time step 2^n, where n is the length of the input. To this end, any halting transition is replaced by a procedure that establishes a binary counter on an extra track of the tape. The counter is increased successively. When it overflows, LBA M has counted up to 2^n and halts. The properties emptiness, finiteness, and infiniteness are not affected by this modification. Moreover, VALC$_{ex}(M)$ is empty, infinite or finite if and only if the language accepted by LBA M is. Therefore, since VALC$_{ex}(M)$ is accepted by some DPWK($\log(n) \cdot \log\log(n)$), the non-semidecidability for LBAs implies the assertion.

Let k be the number of steps that are performed by the unmodified LBA. Then the modified LBA runs through $2^n + k$ configurations each of length n. Therefore $n(2^n + k)$ communications are necessary to accept the corresponding valid computation from VALC(M) and, thus, from VALC$_{ex}(M)$. On the other hand, the length of the valid computation from VALC$_{ex}(M)$ is at least $n2^{2^n+k}$. Since $\log(n2^{2^n+k}) > 2^n + k$ and $\log\log(n2^{2^n+k}) > n$, the number $n(2^n + k)$ of communications allowed exceeds the number of communications necessary, which completes the construction and the proof. □

By reduction of the non-semidecidable problems shown in the previous theorem, we obtain further non-semidecidable properties.

Theorem 9. *The problems of testing inclusion, equivalence, regularity, and context-freeness are not semidecidable for DPWK*$(\log(n) \cdot \log\log(n))$.

Lemma 2 revealed that even two communications suffice to accept non-context-free languages. Moreover, with one single communication non-regular languages can be accepted. However, next we cross the border between decidability and undecidability by considering DPWKs whose communication is bounded by arbitrary constants. In order to prove the results we establish a simulation of a given DPWK$(O(1))$ by reversal-bounded two-way multi-counter machines.

Basically, a *two-way k-counter machine* is a device with a finite-state control, a two-way read-only input head which operates on an input tape delimited by endmarkers, and k counters, each capable of storing any nonnegative integer. At the start of the computation the device is set to a specified initial state with the input head on the left endmarker and all counters set to 0. A move consists of moving the input head a position to the right, to the left, or to keep it at its current position, adding -1, 0, or $+1$ to each counter, and changing the state. The machine can test the counters for zero. The input is accepted if the device eventually halts in an accepting state. A two-way k-counter machine is said to be (r, s)*-reversal bounded* if there are nonnegative integers r and s so that in every accepting computation the input head reverses its direction at most r times and the content in each counter alternately increases and decreases by at most s times. In [5] these machines are formally defined and studied. In particular, it is shown that many properties are decidable and, thus, reversal-bounded two-way multi-counter machines are of tangible advantage for our purposes.

Theorem 10. *For all $k \geq 0$ there are constants r and s such that any DPWK(k) can effectively be simulated by a deterministic (r, s)-reversal bounded two-way 4-counter machine.*

Proof. Let $\mathcal{A}$ be a DPWK(k). We construct an equivalent deterministic two-way 4-counter machine M. Three of the counters, say pos_1, pos_2, and pos_3 are used to position the head of M to the current positions of the components of $\mathcal{A}$. The fourth counter, say run, is used to count a number of steps during the simulations of the components.

The simulation is implemented in at most $k + 1$ phases, where a phase ends when a component communicates or halts. Assume that at the beginning of a phase the head (of M) is on the left endmarker, counter run is 0, counter pos_i contains the current position from the left of the upper component A_1, counter pos_j contains the current position from the right of the lower component A_2, and M has stored the current states of the components in its finite control. This configuration is given at the outset of the simulation and, thus, at the beginning of the first phase. During a phase the following tasks are performed.

1. The head is moved to the current position of component A_1. This can be done by decreasing counter pos_i to 0, whereby the head is moved to the right. In addition, the content of pos_i is successively copied to counter pos_k.

2. The component A_1 is directly simulated until it communicates, halts, or loops without moving (which can be detected by the finite control). In each simulation step the counter *run* is increased by 1.
3. The head is moved to the right endmarker. Then it is moved to the left to the current position of component A_2 with the help of counter pos_j. The content of pos_j is successively copied to counter pos_i.
4. Now component A_2 is directly simulated. If the preceding simulation of A_1 ends with A_1 looping, then A_2 is simulated until it communicates, halts, or loops without moving, and counter *run* is decreased to 0. Else A_2 is simulated whereby the counter *run* is decreased in every step. In this case the simulation stops when A_2 communicates, halts, loops without moving, or when *run* is decreased to 0.

   If M detects that both components run through loops, it rejects the input. If exactly one of the components loops, the other one is considered for the next task. If none of the components loops, the component with fewer simulation steps is considered for the next task. The component can be determined by inspecting counter *run*. If the simulation of A_2 stops due to an empty counter, A_1 is considered, otherwise A_2.
5. The head is moved to the left endmarker if A_1 is considered, otherwise to the right endmarker. Then it is moved again to the current position of the component considered, whereby the corresponding position counter is copied to pos_j.
6. Then the previous simulation of the component considered is restarted, whereby the corresponding position counter is kept up-to-date. Again, the number of simulation steps performed is stored in counter *run*. If the simulation stops with the component communicating, M remembers the message sent.
7. The head is moved to the opposite endmarker in order to initialize the restart of the simulation of the other component. Again, the head is positioned with the help of the position counter and the position counter which is currently empty. Then the simulation is performed for the number of steps given by the content of counter *run*, that is, the number of steps the first component has been simulated.
8. Finally, if one of the components halted, M can decide whether $\mathcal{A}$ accepts or rejects and can do the same. If the simulation ended by a communication, M can internally update the states of A_1 and A_2, move the head back on the left endmarker, and start a new phase.

The effective construction shows that M accepts the language $L(\mathcal{A})$ with four counters. Since in every task at most a constant number of head and counter reversals are performed, it suffices to add these constants in order to compute the numbers r and s. This shows that M is reversal-bounded and concludes the proof. □

It has been shown in [5] that the properties of the following theorem are decidable for deterministic reversal-bounded two-way finite-counter machines. Due to the effectiveness in the construction of Theorem 10 they are decidable for DPWK(k), too.

Theorem 11. *Let $k \geq 0$ be a constant. Then emptiness, finiteness, inclusion, and equivalence are decidable for DPWK(k).*

Another result in [5] says that any language which is accepted by a deterministic reversal-bounded two-way finite-counter machine has a semilinear Parikh image. Thus, the next corollary separates the language classes $\mathscr{L}(\mathrm{DPWK}(O(1)))$ and $\mathscr{L}(\mathrm{DPWK}(O(\log(n))))$ by the witness language L_{expo}.

Corollary 12. *Let $k \geq 0$ be a constant. Every unary language accepted by some DPWK(k) is regular. The languages $\{ a^n b^{2^n} \mid n \geq 0 \}$, $\{ a^n b^{n^2} \mid n \geq 0 \}$ etc. cannot be accepted by any DPWK(k). The languages L_{poly} and L_{expo} cannot be accepted by any DPWK(k).*

References

1. Czeizler, E., Czeizler, E.: On the power of parallel communicating Watson-Crick automata systems. Theoret. Comput. Sci. 358, 142–147 (2006)
2. Czeizler, E., Czeizler, E., Kari, L., Salomaa, K.: On the descriptional complexity of Watson-Crick automata. Theoret. Comput. Sci. 410, 3250–3260 (2009)
3. Freund, R., Păun, G., Rozenberg, G., Salomaa, A.: Watson-Crick finite automata. In: DIMACS Workshop on DNA Based Computers, University of Pennsylvania, pp. 305–317 (1997)
4. Hartmanis, J.: Context-free languages and Turing machine computations. Proc. Symposia in Applied Mathematics 19, 42–51 (1967)
5. Ibarra, O.H.: Reversal-bounded multicounter machines and their decision problems. J. ACM 25, 116–133 (1978)
6. Jurdziński, T., Kutyłowski, M.: Communication gap for finite memory devices. In: Yu, Y., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 1052–1064. Springer, Heidelberg (2001)
7. Jurdziński, T., Kutyłowski, M., Loryś, K.: Multi-party finite computations. In: Asano, T., Imai, H., Lee, D.T., Nakano, S.-i., Tokuyama, T. (eds.) COCOON 1999. LNCS, vol. 1627, pp. 318–329. Springer, Heidelberg (1999)
8. Leupold, P., Nagy, B.: $5' \to 3'$ Watson-Crick automata with several runs. In: Non-Classical Models of Automata and Applications (NCMA 2009). books@ocg.at, vol. 256, pp. 167–180. Austrian Computer Society (2009)
9. Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and Its Applications. Springer, Heidelberg (1993)
10. Martín-Vide, C., Mateescu, A., Mitrana, V.: Parallel finite automata systems communicating by states. Int. J. Found. Comput. Sci. 13, 733–749 (2002)
11. Mitrana, V.: On the degree of communication in parallel communicating finite automata systems. J. Autom. Lang. Comb. 5, 301–314 (2000)
12. Nagy, B.: On $5' \to 3'$ sensing Watson-Crick finite automata. In: Garzon, M.H., Yan, H. (eds.) DNA 2007. LNCS, vol. 4848, pp. 256–262. Springer, Heidelberg (2008)
13. Păun, G., Rozenberg, G., Salomaa, A.: DNA Computing: New Computing Paradigms. Texts in Theoretical Computer Science. Springer, Heidelberg (1998)

Better Hyper-minimization
Not as Fast, But Fewer Errors

Andreas Maletti*

Departament de Filologies Romàniques, Universitat Rovira i Virgili
Avinguda de Catalunya 35, 43002 Tarragona, Spain
andreas.maletti@urv.cat

Abstract. Hyper-minimization aims to compute a minimal deterministic finite automaton (DFA) that recognizes the same language as a given DFA up to a finite number of errors. Algorithms for hyper-minimization that run in time $O(n \log n)$, where n is the number of states of the given DFA, have been reported recently in [Gawrychowski and Jeż: Hyper-minimisation made efficient. Proc. Mfcs, Lncs 5734, 2009] and [Holzer and Maletti: An $n \log n$ algorithm for hyper-minimizing a (minimized) deterministic automaton. *Theor. Comput. Sci.* 411, 2010]. These algorithms are improved to return a hyper-minimal DFA that commits the least number of errors. This closes another open problem of [Badr, Geffert, and Shipman: Hyper-minimizing minimized deterministic finite state automata. Rairo *Theor. Inf. Appl.* 43, 2009]. Unfortunately, the time complexity for the obtained algorithm increases to $O(n^2)$.

1 Introduction

Although nondeterministic and deterministic finite automata (NFA and DFA, respectively) are equally expressive [15], NFA can be exponentially smaller than DFA [12, 14], where the size is measured by the number of states. NFA and DFA are used in a vast number of applications that require huge automata like speech processing [13] or linguistic analysis [11]. Consequently, minimization of automata was studied early on. The minimization problem for DFA (NFA) is the computation of an equivalent DFA (NFA) that has the minimal size (i.e., number of states) of all equivalent DFA (NFA). On the bright side, it was shown that DFA can be efficiently minimized in time $O(n \log n)$ [9], where n is the size of the given DFA. However, minimization for NFA is Pspace-complete [10] and thus impractical.

Here we focus on DFA. Although, they can be efficiently minimized, it is often desirable (or even necessary) to sacrifice correctness to minimize further. This leads to the area of lossy compression, in which certain errors are tolerated in order to allow even smaller DFA. A particularly simple error profile is studied in hyper-minimization [1–3, 5–7], where any finite number of errors is allowed. The algorithms of [5–7] run in time $O(n \log n)$, and thus, are asymptotically as efficient as classical minimization. Given a DFA M, they both return a DFA that

* The author was supported by the *Ministerio de Educación y Ciencia* (MEC) grants JDCI-2007-760 and MTM-2007-63422.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 201–210, 2011.

– recognizes the same language as M up to a finite number of errors, and
– is minimal among all DFA with the former property (hyper-minimal).

Further, Gawrychowski and Jeż [5] report an algorithm that disallows errors on strings exceeding a specified length. This restriction yields a slightly stricter error profile, but their minimization algorithm still runs in time $O(n \log n)$.

In this paper, we extend the basic hyper-minimization algorithms such that, in addition, the returned DFA commits the least number of errors among all DFA with the two, already mentioned properties. A DFA with those three properties is called 'hyper-optimal'. Note that hyper-optimality depends on the input DFA (or better: its recognized language). Moreover, we return the number of committed errors as a quality measure. It allows a user to disregard the returned DFA if the number of errors is unacceptably large. Our result is based essentially on a syntactic characterization [3, Theorems 3.8 and 3.9] of hyper-minimal DFA. Two DFA are almost-equivalent if their recognized languages differ on only finitely many strings (note that this corresponds to the first item mentioned earlier). A preamble state is a state that can be reached by only finitely many strings from the initial state of the DFA. All remaining states are kernel states. The characterization [3, Theorems 3.8 and 3.9] states that the kernels (i.e., the part of the automaton consisting of the kernel states) of all hyper-minimal, almost-equivalent DFA are isomorphic. Moreover, the preambles are almost-isomorphic, which means that they are isomorphic up to the finality of the states. This yields, as already pointed out in [3], that two hyper-minimal, almost-equivalent DFA differ in only three aspects: (i) the finality of preamble states, (ii) the transitions from preamble states to kernel states, and (iii) the initial state. Thus, the characterization allows us to easily consider all hyper-minimal, almost-equivalent DFA to find a hyper-optimal one. We thus solve an open problem stated in [3]. Unfortunately, the time complexity for the obtained algorithm is $O(n^2)$. Whether it can be improved to $O(n \log n)$ remains an open problem.

2 Preliminaries

The integers and nonnegative integers are denoted by $\mathbb{Z}$ and $\mathbb{N}$, respectively. If the symmetric difference $(S \setminus T) \cup (T \setminus S)$ is finite, then S and T are almost-equal. For finite sets Σ, also called alphabets, the set of all strings over Σ is Σ^*, of which the empty string is $\varepsilon \in \Sigma^*$. Concatenation of strings is denoted by juxtaposition and the length of the word $w \in \Sigma^*$ is $|w|$. A language L over Σ is a subset of Σ^*. A deterministic finite automaton (for short: DFA) is a tuple $M = (Q, \Sigma, q_0, \delta, F)$, in which Q is a finite set of states, Σ is an alphabet of input symbols, $q_0 \in Q$ is an initial state, $\delta \colon Q \times \Sigma \to Q$ is a transition function, and $F \subseteq Q$ is a set of final states. The transition function δ extends to a mapping $\delta \colon Q \times \Sigma^* \to Q$ as follows: $\delta(q, \varepsilon) = q$ and $\delta(q, \sigma w) = \delta(\delta(q, \sigma), w)$ for every $q \in Q$, $\sigma \in \Sigma$, and $w \in \Sigma^*$. For every $q \in Q$, let $L(M)_q = \{w \in \Sigma^* \mid \delta(q_0, w) = q\}$. The DFA M recognizes the language $L(M) = \bigcup_{q \in F} L(M)_q$.

Two states $p, q \in Q$ are equivalent, denoted by $p \equiv q$, if $\delta(p, w) \in F$ if and only if $\delta(q, w) \in F$ for every word $w \in \Sigma^*$. The DFA M is minimal if it does not

Algorithm 1. Structure of the hyper-minimization algorithm [6, 7]

Require: a DFA M

$M \leftarrow \text{MINIMIZE}(M)$ // HOPCROFT's algorithm; $O(m \log n)$
2. $K \leftarrow \text{COMPUTEKERNEL}(M)$ // compute the kernel states; $O(m)$
$\sim \leftarrow \text{AEQUIVALENTSTATES}(M)$ // compute almost-equivalence; $O(m \log n)$
4. $M \leftarrow \text{MERGESTATES}(M, K, \sim)$ // merge almost-equivalent states; $O(m)$
return M

have equivalent states (i.e., $p \equiv q$ implies $p = q$). The name 'minimal' is justified by the fact that no DFA with (strictly) fewer states recognizes the same language as a minimal DFA. For every DFA $M = (Q, \Sigma, q_0, \delta, F)$ an equivalent minimal DFA can be computed efficiently using HOPCROFT's algorithm [8], which runs in time $O(m \log n)$ where $m = |Q \times \Sigma|$ and $n = |Q|$.

3 Hyper-minimization

Let us quickly recall hyper-minimization from [1–3, 6, 7]. We will follow the presentation of [6, 7]. Hyper-minimization is a form of lossy compression with the goal of compressing minimal DFA further at the expense of a finite number of errors. Two DFA M_1 and M_2 such that $L(M_1)$ and $L(M_2)$ are almost-equal are *almost-equivalent*. Moreover, a DFA M that admits no almost-equivalent DFA with (strictly) fewer states is *hyper-minimal*. Consequently, hyper-minimization [1–3] aims to find an almost-equivalent, hyper-minimal DFA.

In the following, let $M = (Q, \Sigma, q_0, \delta, F)$ be a minimal DFA. Let $m = |Q \times \Sigma|$ be the number of its transitions and $n = |Q|$ be the number of its states.

Definition 1 (cf. [3, Definition 2.2]). *Two states $p, q \in Q$ are* k-equivalent *with $k \in \mathbb{N}$, denoted by $p \sim_k q$, if $\delta(p, w) = \delta(q, w)$ for every $w \in \Sigma^*$ such that $|w| \geq k$. The* almost-equivalence $\sim \subseteq Q \times Q$ *is $\sim = \bigcap_{k \in \mathbb{N}} \sim_k$.*

Both k- and almost-equivalence are equivalence relations. The set $\text{Pre}(M)$ of *preamble states* is $\{q \in Q \mid L(M)_q \text{ is finite}\}$, and $\text{Ker}(M) = Q \setminus \text{Pre}(M)$ is the set of *kernel states*. The contributions [5–7] report hyper-minimization algorithms that run in time $O(m \log n)$. The overall structure of the hyper-minimization algorithm [6, 7] is displayed in Algorithm 1, and MERGESTATES is displayed in Algorithm 2. The merge of $p \in Q$ into $q \in Q$ redirects all incoming transitions of p to q. If $p = q_0$ then q is the new initial state. The finality of q is not changed even if p is final. Clearly, the state p can be deleted after the merge if $p \neq q$.

Theorem 2 ([3, Section 4] and [7, Theorem 13]). *In time $O(m \log n)$ Algorithm 1 returns a hyper-minimal* DFA *that is almost-equivalent to M.*

4 An Example

In this section, we illustrate the problem that we address in this contribution. Namely, we propose an algorithm that not only returns a hyper-minimal,

Algorithm 2. MERGESTATES: Merge almost-equivalent states [6, 7]

Require: a minimal DFA M, its kernel states K, and its almost-equivalent states $\sim$

for all $B \in (Q/\sim)$ **do**
2. select $q \in B$ with $q \in K$ if $B \cap K \neq \emptyset$ // select $q \in B$, preferably a kernel state
for all $p \in B \setminus K$ **do**
4. merge p into q // merge all preamble states of the block into q
return M

almost-equivalent DFA, but rather one that commits the minimal number of errors among all hyper-minimal, almost-equivalent DFA. Moreover, we return the exact number of errors, and we could also return the error strings (at the expense of an increased run-time). We thus solve an open problem of [3].

Throughout this section, we consider the minimal DFA M of Fig. 1, which is essentially the minimal DFA of [3, Fig. 2] with two new states 0 and 2. We added those states because all hyper-minimal DFA that are almost-equivalent to the original DFA of [3] commit exactly 9 errors. Consequently, the existing algorithms already yield DFA with the minimal number of errors. The two new states 0 and 2, of which 0 is the new initial state, change the situation.

The kernel states of M are $\mathrm{Ker}(M) = \{E, F, I, J, L, M, P, Q, R\}$ and the almost-equivalence (represented as a partition) is

$$\{\{0\}, \{2\}, \{A\}, \{B\}, \{C, D\}, \{E\}, \{F\}, \{G, H, I, J\}, \{L, M\}, \{P, Q\}, \{R\}\} \ .$$

Both $\mathrm{Ker}(M)$ and $\sim$ can be computed with the existing algorithms of [2,3,5–7]. The hyper-minimization algorithm of [6, 7] might return the hyper-minimal DFA M_1 of Fig. 2 (left), which is almost-equivalent to M. Another such hyper-minimal, almost-equivalent DFA M_2 is presented in Fig. 2 (right). To the author's knowledge there is no hyper-minimization algorithm that can produce M_2. All known algorithms merge both G and H into one of the almost-equivalent kernel states I and J (see Algorithm 2). For example, M_1 is obtained by merging G and H into I. However, M_2 is obtained by merging H into J and G into I. Now, let us look at the errors that M_1 and M_2 make in comparison to M. The following display lists those errors for M_1 (left) and M_2 (right), of which the specific errors of one of the two DFA are underlined.

$$\begin{aligned}
&\{\underline{aa}, aaa, \underline{aaaaa}, aaabaa,\\
&\ aabb, aabbbaa, abb, abbbaa,\\
&\ babb, babbaa, babbbaa, \underline{bbaaa},\\
&\ \underline{bb}, bba, bbabaa, bbbb, bbbbbaa\}
\end{aligned}
\qquad
\begin{aligned}
&\{aaa, aaabaa\\
&\ aabb, aabbbaa, abb, abbbaa,\\
&\ \underline{bab}, \underline{babaaa}, babb, babbaa, babbbaa,\\
&\ bba, bbabaa, bbbb, bbbbbaa\}
\end{aligned}$$

We observe that M_1 commits 17 errors, whereas M_2 commits only 15 errors. Consequently, there is a qualitative difference in different hyper-minimal, almost-equivalent DFA. To be more precise, the quality of the obtained hyper-minimal DFA depends significantly on how the merges are performed.

Let us take a closer look at the cause of the errors. Since the final state C is merged into the non-final state D to obtain M_1, the combined state D of M_1 is

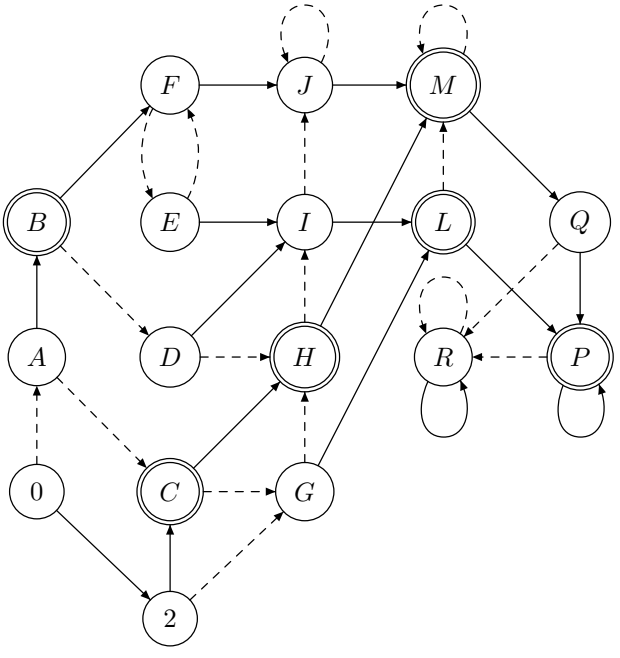


Fig. 1. An example DFA with a-transitions (straight lines) and b-transitions (dashed lines). The initial state is 0.

non-final. Consequently, all strings of $L(M)_C = \{aa, bb\}$, which were accepted in M, are now rejected in M_1. Analogously, the error bab of M_2 is caused by the merge of D into C. The number of those errors can easily be computed with a folklore algorithm (see Algorithm 3 and [4, Lemma 4]) that computes the number of paths from q_0 to each preamble state. Mind that the graph of a DFA restricted to its preamble states is acyclic.

Theorem 3 (see [4, Lemma 4]). *Algorithm 3 computes the number of paths to each preamble state in time $O(m)$.*

Example 4. Algorithm 3 computes the following for M.

$$w(0) = w(2) = w(A) = w(B) = w(D) = 1 \quad w(C) = 2 \quad w(G) = 3 \quad w(H) = 6$$

The remaining errors of M_1 are caused by the merges of G and H into the almost-equivalent kernel state I. Let us denote by $E_{p,q}$ the number of errors made between almost-equivalent states $p \sim q$. More formally, this is the number of strings in the symmetric difference of $L(M_p)$ and $L(M_q)$, where for every $q' \in Q$, the DFA $M_{q'}$ is $(Q, \Sigma, q', \delta, F)$. In other words, $M_{q'}$ is the same DFA as M with initial state q'. Clearly, $E_{q,q} = 0$ and $E_{p,q} = E_{q,p}$ for every $p, q \in Q$. For example, $E_{G,I} = 2$ and $E_{H,I} = 3$ and the corresponding error strings are $\{b, bbaa\}$ and $\{\varepsilon, aa, baa\}$, respectively. Actually, we only need to consider transitions of M_1 that connect preamble to kernel states due to a characterization result of [3]. For

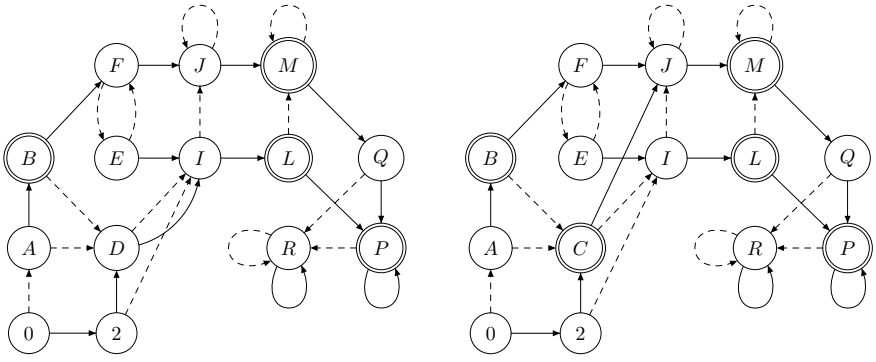


Fig. 2. Two resulting hyper-minimal DFA with a-transitions (straight lines) and b-transitions (dashed lines). The initial state is 0 in both cases.

Algorithm 3. CompAccess: Compute the number of paths to preamble states

Require: a minimal DFA $M = (Q, \Sigma, q_0, \delta, F)$, its preamble states P, and a topological sorting $o\colon \mathbb{N} \to P$ of the preamble states

$w(o(0)) \leftarrow 1$ // the path ε leads to $q_0 = o(0)$

2. **for** $i = 1$ to $|P|$ **do**

$w(o(i)) \leftarrow \sum_{\substack{q \in Q, \sigma \in \Sigma \\ \delta(q,\sigma) = o(i)}} w(q)$ // for each transition (q, σ) leading to $o(i)$ add $w(q)$

4. **return** w

example, for the transition $D \xrightarrow{a} I$ of M_1, we first identify the states of M that were merged into D of M_1. These are C and D of M. Next, we compute the number of paths (in M) to them for each such state q and multiply it with the number of errors made between $\delta(q, a)$ and I. The such obtained error counts are summed up for the total error count. For the three relevant transitions in M_1 we obtain:

$D \xrightarrow{a} I$	$D \overset{b}{\dashrightarrow} I$	$2 \overset{b}{\dashrightarrow} I$
$w(C) \cdot E_{\delta(C,a),I} = 6$	$w(C) \cdot E_{\delta(C,b),I} = 4$	$w(2) \cdot E_{\delta(2,b),I} = 2$
$w(D) \cdot E_{\delta(D,a),I} = 0$	$w(D) \cdot E_{\delta(D,b),I} = 3$	
Sum $= 6$	Sum $= 7$	Sum $= 2$

Thus, we identified $6 + 7 + 2 = 15$ errors. Together with the 2 errors that were caused by the non-finality of D we obtained all 17 errors committed by M_1.

5 Optimal State Merging

The approach presented in the previous section suggests how to compute a hyper-optimal DFA for a given minimal DFA $M = (Q, \Sigma, q_0, \delta, F)$ with $m = |Q \times \Sigma|$ and $n = |Q|$. We can simply compute the number of errors for all hyper-minimal, almost-equivalent DFA and select a DFA with a minimal error count. We have

Algorithm 4. CompErrors: Compute the number of errors made between almost-equivalent states

Require: minimal DFA $M = (Q, \Sigma, q_0, \delta, F)$ and states $p \sim q$
Global: error matrix $E \in \mathbb{Z}^{Q \times Q}$ initially 0 on the diagonal and -1 everywhere else

if $E_{p,q} \neq -1$ **then**
2. **return** $E_{p,q}$ // if already computed, then return stored value
$c \leftarrow ((p \in F) \text{ xor } (q \in F))$ // set errors to 1 if p and q differ on finality
4. $E_{p,q} \leftarrow c + \sum_{\sigma \in \Sigma} \text{CompErrors}(M, \delta(p,\sigma), \delta(q,\sigma))$ // add errors from follow-states
return $E_{p,q}$ // return the computed value

already seen that different parts (finality and merges) are responsible for the errors. The different parts do not affect each other, which yields that we can compute the number of errors for each choice and greedily select the best one.

First, let us address how to compute the values $E_{p,q}$ for $p \sim q$. Our algorithm is presented in Algorithm 4. It inspects the global matrix E whether the value was already computed. If not, then it checks whether p and q differ on finality (i.e., whether ε is in the symmetric difference of $L(M_p)$ and $L(M_q)$) and adds the error counts $E_{\delta(p,\sigma),\delta(q,\sigma)}$ for each $\sigma \in \Sigma$.

Theorem 5. *Algorithm 4 computes all $E_{p,q}$ with $p \sim q$ in time $O(mn)$.*

Proof. Clearly, the initialization and the recursion for $E_{p,q}$ are correct because each error string w is either the empty string ε or it starts with a letter $\sigma \in \Sigma$. In the latter case, $w' \in \Sigma^*$ with $w = \sigma w'$ is an error string for $E_{\delta(p,\sigma),\delta(q,\sigma)}$. For every $p \sim q$ there exists $k \in \mathbb{N}$ such that $p \sim_k q$ and thus $\delta(p, w) = \delta(q, w)$ for every $w \in \Sigma^*$ with $|w| \geq k$. Consequently, at most k nested recursive calls can occur in the computation of $E_{p,q}$, which proves that the recursion terminates. It remains to prove the time bound. Obviously, if $E_{p,q}$ was already computed, then the algorithm returns immediately. Thus, it makes at most n^2 calls because then all values $E_{p,q}$ are computed. Moreover, there are at most $|\Sigma| + 1$ summands in line 7. Consequently, each call executes in time $O(|\Sigma|)$ apart from the recursive calls, which yields that the algorithm runs in time $O(|\Sigma|n^2) = O(mn)$. □

Example 6. Let us illustrate Algorithm 4 on the example DFA of Fig. 1 and the almost-equivalence $\sim$. We list some error matrix entries together with the corresponding error strings. Note that the error strings are not computed by the algorithm, but are presented for illustrative purposes only.

$E_{Q,P} = 1$	$\{\varepsilon\}$	$E_{H,J} = 2$	$\{\varepsilon, baa\}$	$E_{G,J} = 3$	$\{aa, b, bbaa\}$
$E_{L,M} = 1$	$\{a\}$	$E_{H,I} = 3$	$\{\varepsilon, aa, baa\}$	$E_{G,I} = 2$	$\{b, bbaa\}$
$E_{I,J} = 1$	$\{aa\}$			$E_{G,H} = 5$	$\{\varepsilon, aa, b, baa, bbaa\}$

Next, we need to shortly discuss the structural similarities between hyper-minimal, almost-equivalent DFA. It was shown in [3, Theorems 3.8 and 3.9] that two such DFA

Algorithm 5. CompFinality: Determine finality of a block of preamble states

Require: a minimal DFA $M = (Q, \Sigma, q_0, \delta, F)$, a block of preamble states B, and the number $w(p)$ of access paths for each preamble state p

Global: error count e

$(\overline{f}, f) \leftarrow \Big(\sum_{q \in B \cap F} w(q), \sum_{q \in B \setminus F} w(q) \Big)$ // errors for non-final and final state

2. $e \leftarrow e + \min(\overline{f}, f)$ // add smaller value to global error count

select $q \in B$ such that $q \in F$ if $\overline{f} > f$ // select final state if fewer errors for finality

4. **return** q // return selected state

have isomorphic kernels and almost-isomorphic (by an isomorphism not necessarily respecting finality) preambles. This yields that those DFA only differ on three aspects, which were already identified in [3]:

- the finality of preamble states,
- transitions from preamble states to kernel states, and
- the initial state.

All of the following algorithms will use a global variable e, which will keep track of the number of errors. Initially, it will be set to 0 and each discovered error will increase it. First, we discuss ComputeFinality. For the given block B of almost-equivalent preamble states it computes the number of access paths to final and non-final states in B. Each such path represents a string of $\bigcup_{q \in B} L(M)_q$. After the merge all those strings will take the hyper-minimal DFA into the same state. Thus, making this state final, will cause the number f of errors computed in the algorithm because each access path to a non-final state of B will now access a final state after the merge.

Lemma 7. ComputeFinality(M, B, w) *adds the number of errors made in state p when merging all states of the block B of almost-equivalent preamble states into the state p that is returned by the call.*

Next, we discuss the full merging algorithm (see Algorithm 6). We assume that all values $E_{p,q}$ with $p \sim q$ are already computed. In lines 5–7 we first handle the already discussed decision for the finality of blocks B of preamble states and perform the best merge into state q. In lines 8–11 we investigate the second structural difference between hyper-minimal, almost-equivalent DFA: transitions from preamble to kernel states. Clearly, the preamble state represents a set of exclusively preamble states in the input DFA M and the kernel state represents a set of almost-equivalent states of M that contains at least one kernel state. Consequently, we can simply check whether $\delta(q, \sigma)$ is almost-equivalent to a kernel state. We then consider all almost-equivalent kernel states $q' \sim \delta(q, \sigma)$ and compute the error-count for rerouting the transition to q'. This error count is simply obtained by multiplying the number of paths to a state p in the current block B with the number $E_{\delta(p,\sigma),q'}$ of errors performed between the designated kernel state and the follow-state of the current state. Mind that $\delta(p, \sigma)$ was not

Algorithm 6. OptMerge: Optimal merging of almost-equivalent states

Require: a minimal DFA $M = (Q, \Sigma, q_0, \delta, F)$, its kernel states K, and its almost-equivalent states $\sim$

Global: error count e; initially 0

$P \leftarrow Q \setminus K$ // set P to preamble states
2. $o \leftarrow \text{TopoSort}(P)$ // topological sorting of preamble states; $o\colon \mathbb{N} \to P$
$w \leftarrow \text{CompAccess}(M, P, o)$ // compute the number of access paths for preamble
4. **for all** $B \in (Q/\!\sim)$ such that $B \subseteq P$ **do**
$q \leftarrow \text{CompFinality}(M, B, w)$ // determine finality of merged state
6. **for all** $p \in B$ **do**
merge p into q // perform the merges
8. **for all** $\sigma \in \Sigma$ **do**
if $B' = \{q' \in K \mid q' \sim \delta(q, \sigma)\} \neq \emptyset$ **then**
10. $e \leftarrow e + \min_{q' \in B'} \Big(\sum_{p \in B} w(p) \cdot E_{\delta(p,\sigma),q'} \Big)$ // add best error count
$\delta(q, \sigma) \leftarrow \arg\min_{q' \in B'} \Big(\sum_{p \in B} w(p) \cdot E_{\delta(p,\sigma),q'} \Big)$ // update follow state
12. **if** $B' = \{q' \in K \mid q' \sim q_0\} \neq \emptyset$ **then**
$e \leftarrow e + \min_{q' \in B'} E_{q_0,q'}$ // add best error count
14. $q_0 \leftarrow \arg\min_{q' \in B'} E_{q_0,q'}$ // set best initial state
return (M, e)

affected by the merge in lines 6–7 because the merge only reroutes incoming transitions to p. If there are several states in the current block B, then we sum the obtained error counts. The smallest such error count is then added to the global error count in line 10 and the corresponding designated kernel state is selected as the new target of the transition in line 11. This makes all preamble states that are almost-equivalent to a kernel state unreachable, so they could be removed. Finally, if the initial state is almost-equivalent to a kernel state, then we perform the same steps as previously mentioned to determine the new initial state (i.e., we consider the transition from "nowhere" to the initial state).

A DFA M' is *hyper-optimal for* $L(M)$ if it is hyper-minimal and the cardinality of the symmetric difference between $L(M)$ and $L(M')$ is minimal among all hyper-minimal DFA. Note that a hyper-optimal DFA for $L(M)$ is almost-equivalent to M.

Theorem 8. *Algorithm 6 runs in time $O(mn)$ and returns a hyper-optimal dfa for $L(M)$. In addition, the number of errors committed is returned.*

Proof. The time complexity is easy to check, so we leave it as an exercise. Since the choices (finality, transition target, initial state) are independent, all hyper-minimal, almost-equivalent DFA are considered in Algorithm 6 by [3, Theorems 3.8 and 3.9]. Consequently, we can always select the local optimum for each choice to obtain a global optimum, which proves that the returned number is the

minimal number of errors among all hyper-minimal DFA. Mind that the number of errors would be infinite for a hyper-minimal DFA that is not almost-equivalent to M. Moreover, it is obviously the number of errors committed by the returned DFA, which proves that the returned DFA is hyper-optimal for $L(M)$. □

Corollary 9 (of Theorem 8). *For every* DFA M *we can obtain a hyper-optimal* DFA *for* $L(M)$ *in time* $O(mn)$.

References

1. Badr, A.: Hyper-minimization in $O(n^2)$. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 223–231. Springer, Heidelberg (2008)
2. Badr, A.: Hyper-minimization in $O(n^2)$. Int. J. Found. Comput. Sci. 20(4), 735–746 (2009)
3. Badr, A., Geffert, V., Shipman, I.: Hyper-minimizing minimized deterministic finite state automata. RAIRO Theor. Inf. Appl. 43(1), 69–94 (2009)
4. Eppstein, D.: Finding common ancestors and disjoint paths in DAGs. Tech. Rep. 95-52, University of California, Irvine (1995)
5. Gawrychowski, P., Jeż, A.: Hyper-minimisation made efficient. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 356–368. Springer, Heidelberg (2009)
6. Holzer, M., Maletti, A.: An $n \log n$ algorithm for hyper-minimizing states in a (minimized) deterministic automaton. In: Maneth, S. (ed.) CIAA 2009. LNCS, vol. 5642, pp. 4–13. Springer, Heidelberg (2009)
7. Holzer, M., Maletti, A.: An $n \log n$ algorithm for hyper-minimizing a (minimized) deterministic automaton. Theor. Comput. Sci. 411(38–39), 3404–3413 (2010)
8. Hopcroft, J.E.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Theory of Machines and Computations, pp. 189–196. Academic Press, London (1971)
9. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison Wesley, Reading (2007)
10. Jiang, T., Ravikumar, B.: Minimal NFA problems are hard. SIAM J. Comput. 22(6), 1117–1141 (1993)
11. Johnson, C.D.: Formal Aspects of Phonological Description. Monographs on Linguistic Analysis, vol. 3. Mouton, The Hague (1972)
12. Meyer, A.R., Fischer, M.J.: Economy of description by automata, grammars, and formal systems. In: Proc. 12th IEEE Annual Symp. Switching and Automata Theory, pp. 188–191. IEEE Computer Society Press, Los Alamitos (1971)
13. Mohri, M.: Finite-state transducers in language and speech processing. Comput. Linguist. 23(2), 269–311 (1997)
14. Moore, F.R.: On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. IEEE Trans. Computers 20(10), 1211–1214 (1971)
15. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Res. Dev. 3(2), 115–125 (1959)

Regular Expressions on Average and in the Long Run

Manfred Droste and Ingmar Meinecke

Institut für Informatik, Universität Leipzig,
04109 Leipzig, Germany
{droste,meinecke}@informatik.uni-leipzig.de

Abstract. Quantitative aspects of systems like consumption of resources, output of goods, or reliability can be modeled by weighted automata. Recently, objectives like the average cost or the longtime peak power consumption of a system have been modeled by weighted automata which are not semiring weighted anymore. Instead, operations like limit superior, limit average, or discounting are used to determine the behavior of these automata. Here, we introduce a new class of weight structures subsuming a range of these models as well as semirings. Our main result shows that such weighted automata and Kleene-type regular expressions are expressively equivalent both for finite and infinite words.

1 Introduction

Recently, a new kind of weighted automata was established by Chatterjee, Doyen, and Henzinger [3,4,5,6] which compute objectives like the long-run average cost or long-run maximal reward. These models enrich the automata toolbox for the modeling of quantitative aspects of systems which may be the consumption of some resource like time, power, or memory, or the production of some output like goods or pollution. Objectives like average or longtime peaks cannot be modeled by classical semiring weighted automata [1,7,15,16,20] or lattice automata [17]. Therefore, the theory of semiring weighted automata does not carry over to those new weighted automata.

Finite automata and regular expressions describe the same class of languages [14]. This result by Kleene was transfered by Schützenberger [21] to the semiring-weighted setting over finite words. For infinite words, the respective equivalence for ω-languages was shown by Büchi [2] and for the weighted setting by Ésik and Kuich [12,13] for semiring-semimodule pairs, by Droste and Kuske [8] for discounting, and by Droste and Vogler [10] for bounded lattices. In this paper, we will establish that regular weighted expressions are expressively equivalent to the new kind of weighted automata computing average and longtime behavior.

In the weighted automata considered here, the weight of a run is calculated in a global way by means of a valuation function. For finite runs, examples of valuation functions are the average, the supremum, or the last value. For infinite runs, we may consider the limit superior of the weights, the limit average, i.e., the longrun average weight, or a discounted sum. As usual, non-determinism is resolved by a monoid operation which is written as a sum. Our automata model has features of classical finite automata and of the weighted automata from [3]. The use of a valuation function is due to [3]. But, moreover, we allow acceptance conditions like final states in the case of

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 211–221, 2011.

finite words and a Büchi condition for non-terminating behavior. The computation of the weight of an infinite run is realized by three ingredients: *(i)* the Büchi condition, *(ii)* a valuation function for finite sequences, and *(iii)* an ω-indexed valuation function for infinite sequences. Hereby, the finite sequences of weights between two consecutive acceptance states are evaluated by the valuation function and then these infinitely many intermediate results are combined by the ω-indexed valuation function. This procedure guarantees the necessary link between finite and ω-automata in order to establish a Kleene-like result also for infinite words.

Our main results are as follows. For finite words we show in Theorem 4.2 that weighted automata and regular weighted expressions are expressively equivalent. The weights are taken from Cauchy valuation monoids D which have, besides the sum and the valuation function, a family of products. Such a product is parameterized by two natural numbers representing the length of two words to be concatenated. By these products we can define the Cauchy product and the iteration of functions $S : \Sigma^+ \to D$. Cauchy valuation monoids generalize the valuation functions considered in [3], semirings, and more. For infinite words, we present by Cauchy ω-indexed valuation monoids also a unified setting for the weight structure. They comprise a sum operation, a valuation and an ω-indexed valuation function, as well as a family of parameterized products. Now we have also products where the first parameter is a positive integer but the second one is ω which will be used for the concatenation of a finite and an infinite word. Moreover, we have to impose more restrictions on the interaction of the different operations. This is not surprising because one has to do so also in other settings, cf. [13]. However, instantiations of Cauchy ω-indexed valuation monoids are the structures with limit superior, limit average, or discounting as considered in [3] as well as the complete star-omega-semirings [12,13] or the semirings used in [9,19]. We show that over Cauchy ω-indexed valuation monoids, every ω-regular weighted expression can be translated into an equivalent weighted Büchi automaton, see Theorem 5.4(a). Conversely, under an additional assumption called the partition property, which governs the computation of an infinite sequence by using different partitions of the sequence, the behavior of every weighted Büchi automaton can be described by an ω-regular expression, cf. Theorem 5.4(b).

Our setting for infinite words owes some ideas to the case of discounting [8]. The main difference to the setting of Ésik and Kuich [12,13] is the absence of an infinitary associativity for the ω-indexed valuation function which leads to a range of complications. But we have to drop this property in order to include the new models like limit superior or limit average [3]. Nevertheless, we succeed in proving a Kleene-like result in a unified framework.

2 Weighted Automata on Finite and Infinite Words

Let $\mathbb{N}$ denote the set of positive integers and let Σ be an alphabet. By Σ^+ we denote the set of non-empty finite words and by Σ^ω the set of infinite words.

A monoid $(D, +, \mathbb{0})$ is *complete* [11] if it has infinitary sum operations $\sum_I : D^I \to D$ for any index set I such that $\sum_{i\in\emptyset} d_i = \mathbb{0}$, $\sum_{i\in\{k\}} d_i = d_k$, $\sum_{i\in\{j,k\}} d_i = d_j + d_k$ for $j \neq k$, $\sum_{j\in J}\left(\sum_{i\in I_j} d_i\right) = \sum_{i\in I} d_i$ if $\bigcup_{j\in J} I_j = I$ and $I_j \cap I_k = \emptyset$ for $j \neq k$.

For a set D, let $(\mathbb{N} \times D)^\omega = \{(n_i, d_i)_{i\in\mathbb{N}} \mid \forall i \in \mathbb{N} : n_i \in \mathbb{N}, d_i \in D\}$.

Definition 2.1. *A* valuation monoid $(D, +, \mathrm{val}, \mathbb{0})$ *consists of a commutative monoid* $(D, +, \mathbb{0})$ *and a* valuation function $\mathrm{val} : D^+ \to D$ *with* $\mathrm{val}(d) = d$ *and* $\mathrm{val}(d_1, \ldots, d_n) = \mathbb{0}$ *whenever* $d_i = \mathbb{0}$ *for some* $i \in \{1, \ldots, n\}$.

An ω-indexed valuation monoid $(D, +, \mathrm{val}, \mathrm{val}^\omega, \mathbb{0})$ *is a complete valuation monoid* $(D, +, \mathrm{val}, \mathbb{0})$ *equipped with an* ω-indexed valuation function $\mathrm{val}^\omega : (\mathbb{N} \times D)^\omega \to D$ *such that* $\mathrm{val}^\omega(n_k, d_k)_{k\in\mathbb{N}} = \mathbb{0}$ *whenever* $d_k = \mathbb{0}$ *for some* $k \in \mathbb{N}$.

Definition 2.2. *A* weighted (finite) automaton $\mathcal{A} = (Q, I, T, F, \gamma)$ *over the alphabet* Σ *and a valuation monoid* $(D, +, \mathrm{val}, \mathbb{0})$ *consists of a finite state set* Q*, a set* $I \subseteq Q$ *of initial states, a set* $F \subseteq Q$ *of final states, a set* $T \subseteq Q \times \Sigma \times Q$ *of transitions, and a weight function* $\gamma : T \to D$.

A weighted Büchi automaton $\mathcal{A} = (Q, I, T, F, \gamma)$ *over the alphabet* Σ *and an* ω*-indexed valuation monoid* $(D, +, \mathrm{val}, \mathrm{val}^\omega, \mathbb{0})$ *is defined as a weighted finite automaton.*

A weighted automaton is a usual finite automaton equipped with weights for the transitions. Moreover, the automaton can be assumed to be *total* as in [3], i.e., for every $q \in Q$ and every $a \in \Sigma$ there is some $q' \in Q$ with $(q, a, q') \in T$, by adding transitions with weight $\mathbb{0}$. *Runs* $R = (t_i)_{1\le i\le n}$ are defined as finite sequences of matching transitions $t_i = (q_{i-1}, a_i, q_i)$ where $|R| = n$ is the *length* of R. We call the word $w = \ell(R) = a_1 a_2 \ldots a_n$ the *label* of the run R and R a run on w. Moreover, $\gamma(R) = \big(\gamma(t_i)\big)_{1\le i\le n}$ is the sequence of the transition weights of R and $\mathrm{wgt}(R) = \mathrm{val}(\gamma(R))$ is the *weight* of R. A run is *successful* if it starts in an initial state from I and ends in a final state from F. We denote the set of successful runs of $\mathcal{A}$ by $\mathrm{succ}(\mathcal{A})$.

The *behavior* of $\mathcal{A}$ is the function $\|\mathcal{A}\| : \Sigma^+ \to D$, defined by $\|\mathcal{A}\|(w) = \sum\big(\mathrm{val}(\gamma(R)) \mid R \in \mathrm{succ}(\mathcal{A}) \text{ and } \ell(R) = w\big)$ for $w \in \Sigma^+$; if there is no successful run on w, then $\|\mathcal{A}\|(w) = \mathbb{0}$. Any function $S : \Sigma^+ \to D$ is called a *series* (or a *quantitative language* as in [3]) over Σ^+. If S is the behavior of some weighted automaton, then S is called *recognizable*. For reasons of technical simplicity, we do not consider initial or final weights of a weighted automaton and, thus, also not the empty word ε.

Similarly, we define the behavior of weighted Büchi automata $\mathcal{A} = (Q, I, T, F, \gamma)$. Now a *run* $R = (t_i)_{i\in\mathbb{N}}$ is an infinite sequence of matching transitions $t_i = (q_{i-1}, a_i, q_i)$ with *label* $w = \ell(R) = a_1 a_2 \cdots \in \Sigma^\omega$. Let $F(R) = \{j \in \mathbb{N} \mid q_j \in F\}$. Note that $F(R)$ can be finite or infinite. If $F(R)$ is infinite, we enumerate $F(R)$ by $j_1 < j_2 < j_3 < \ldots$ and put $j_0 = 0$. Let $R_k = (t_i)_{j_{k-1}\le i<j_k}$ be the finite sub-run of R starting in $q_{j_{k-1}}$ and terminating in the k-th acceptance state q_{j_k}. Let $\gamma(R_k) = (\gamma(t_i))_{j_{k-1}\le i<j_k}$ be the finite sequence of weights from R_k. Now the *weight of* R is defined as $\mathrm{wgt}(R) = \mathrm{val}^\omega\big(|R_k|, \mathrm{val}(\gamma(R_k))\big)_{k\in\mathbb{N}}$ if $F(R)$ is infinite, and $\mathrm{wgt}(R) = \mathbb{0}$ otherwise. The run R is *successful* if $q_0 \in I$ and $F(R)$ is infinite, i.e., R starts in an initial state and satisfies a Büchi condition with regard to the acceptance set F. The set of successful runs of $\mathcal{A}$ is denoted by $\mathrm{succ}(\mathcal{A})$.

Now the *behavior* of $\mathcal{A}$ is the function $\|\mathcal{A}\| : \Sigma^\omega \to D$ given by $\|\mathcal{A}\|(w) = \sum\big(\mathrm{wgt}(R) \mid R \in \mathrm{succ}(\mathcal{A}) \text{ and } \ell(R) = w\big)$ for $w \in \Sigma^\omega$; if w has no successful run in $\mathcal{A}$, then $\|\mathcal{A}\|(w) = \mathbb{0}$. Any function $S : \Sigma^\omega \to D$ is called an ω-series. S is ω*-recognizable* if there is a weighted Büchi automaton $\mathcal{A}$ with $\|\mathcal{A}\| = S$.

Let $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$, the extended real line.

Example 2.3. $(\mathbb{R}\cup\{-\infty\}, \sup, \mathrm{avg}, -\infty)$ with $\mathrm{avg}(d_1,\ldots,d_n) = \frac{1}{n}\sum_{i=1}^{n} d_i$ is a valuation monoid. A weighted automaton over this valuation monoid takes the arithmetic mean of the weights of the transitions and resolves non-determinism by sup. Moreover, $(\overline{\mathbb{R}}, \sup, \mathrm{avg}, \limsup \mathrm{avg}, -\infty)$ is an ω-indexed valuation monoid when we put for $(n_i, d_i)_{i\in\mathbb{N}} \in (\mathbb{N}\times\overline{\mathbb{R}})^\omega$:

$$\limsup \mathrm{avg}(n_i,d_i)_{i\in\mathbb{N}} = \limsup_k \left(\frac{n_1\cdot d_1+\ldots+n_k\cdot d_k}{n_1+\ldots+n_k}\right)$$

with the exception that $\limsup \mathrm{avg}(n_i,d_i)_{i\in\mathbb{N}} = -\infty$ whenever there is a $j\in\mathbb{N}$ such that $d_i \neq \infty$ for all $i \geq j$ and $\limsup_{k\geq j}\big((n_j d_j+\ldots+n_k d_k)/(n_j+\ldots+n_k)\big) = -\infty$. The weight along an infinite run is the limit superior of the means of the weights of the finite prefixes of the run which end in an acceptance state.

Similarly, $(\mathbb{R}\cup\{\infty\}, \inf, \mathrm{avg}, \infty)$ and $(\overline{\mathbb{R}}, \inf, \mathrm{avg}, \liminf \mathrm{avg}, \infty)$ are a valuation and an ω-indexed valuation monoid, respectively, with a dual definition of $\liminf \mathrm{avg}$.

Example 2.4. $(\overline{\mathbb{R}}, \sup, \sup^{\mathbb{0}}, \limsup, -\infty)$ is an ω-indexed valuation monoid with $\sup^{\mathbb{0}}(d_1,\ldots,d_m) = \sup(d_1,\ldots,d_m)$ if $d_i \neq -\infty$ for all $i = 1,\ldots,m$ and $\sup^{\mathbb{0}}(d_1,\ldots,d_m) = -\infty$ otherwise, and $\limsup(n_i,d_i)_{i\in\mathbb{N}} = \limsup_i(d_i)_{i\in\mathbb{N}}$ if $d_i \neq -\infty$ for all $i\in\mathbb{N}$ and $\limsup(n_i,d_i)_{i\in\mathbb{N}} = -\infty$ otherwise. A weighted Büchi automaton over this structure computes the limit superior of the weights along a run.

Whereas the last ω-indexed valuation monoids were considered in [3], the next one is a variation of the $\limsup$-ω-indexed valuation monoid.

Example 2.5. Consider $(\mathbb{R}\cup\{-\infty\}, \sup, \mathrm{last}, -\infty)$ with $\mathrm{last}(d_1,\ldots,d_n) = d_n$ if $d_i \neq -\infty$ for all $i = 1,\ldots,n$ and $\mathrm{last}(d_1,\ldots,d_n) = -\infty$ otherwise. This structure yields a valuation monoid where the weight of the last transition determines the weight of the whole run. The structure $(\overline{\mathbb{R}}, \sup, \mathrm{last}, \limsup, -\infty)$ yields an ω-indexed valuation monoid where the weight of a run is the limit superior of the weights of the transitions entering acceptance states.

Example 2.6 (discounting [3,8]). Let $\lambda > 0$. Then $(\mathbb{R}\cup\{-\infty\}, \sup, \mathrm{disc}_\lambda, -\infty)$ with $\mathrm{disc}_\lambda(d_0,\ldots,d_n) = \sum_{i=0}^{n}\lambda^i d_i$ is a valuation monoid. Moreover, if $0<\lambda<1$ and $\overline{\mathbb{R}}_+ = \{r\in\mathbb{R} \mid r\geq 0\}\cup\{-\infty,\infty\}$, then the structure $(\overline{\mathbb{R}}_+, \sup, \mathrm{disc}_\lambda, \lim \mathrm{disc}_\lambda, -\infty)$ is an ω-indexed valuation monoid where the ω-indexed valuation function $\lim \mathrm{disc}_\lambda(n_i,d_i)_{i\in\mathbb{N}}$ equals

$$\lim_{k\to\infty}\left(\lambda^0 d_1 + \lambda^{n_1} d_2 + \ldots + \lambda^{n_1+\ldots+n_{k-1}} d_k\right).$$

Remark 2.7. Classical weighted automata are defined over *semirings*, cf. [7]. There, weights are multiplied along a run and summed up over all possible runs. This setting fits into our framework: Let $\mathbb{K} = (K,+,\cdot,\mathbb{0},\mathbb{1})$ be a semiring. Now we define $\mathrm{val}(d_1,\ldots,d_n) = d_1\cdot\ldots\cdot d_n$. Then $(K,+,\mathrm{val},\mathbb{0})$ is a valuation monoid. Examples for such semirings are the natural numbers with addition and multiplication or the reals together with sup as the sum operation and inf, sup, or addition as multiplication.

3 Cauchy Valuation Monoids, Cauchy Products, and Iterations

Our goal is to provide a characterization of the behaviors of weighted automata by means of regular expressions. For this, we need a product modeling the concatenation of runs of weighted automata. For semiring-weighted automata over a semiring $\mathbb{K}$ this can be modeled by the Cauchy product of two formal power series $S, S' : \Sigma^+ \to \mathbb{K}$, defined by $S \cdot S'(w) = \sum_{w=uv} S(u) \cdot S'(v)$. In the setting considered here, such a definition turns out to be more difficult because the valuation function evaluates runs in a global way. Here, we use a parameterized product which considers, besides the values to be multiplied, also two natural numbers representing the lengths of the sequences to be combined.

Definition 3.1. *The structure* $\big(D, +, \mathrm{val}, (\cdot_{m,n} \mid m, n \in \mathbb{N}), \mathbb{0}\big)$ *is a* Cauchy valuation monoid *if* $(D, +, \mathrm{val}, \mathbb{0})$ *is a valuation monoid and* $\cdot_{m,n} : D \times D \to D$ *with* $m, n \in \mathbb{N}$ *is a family of products such that for all* $d, d_i, d'_j \in D$ *and all finite subsets* $A, B \subseteq_{fin} D$*:*

$$\mathbb{0} \cdot_{m,n} d = d \cdot_{m,n} \mathbb{0} = \mathbb{0}, \tag{1}$$

$$\mathrm{val}(d_1, \ldots, d_m, d'_1, \ldots, d'_n) = \mathrm{val}(d_1, \ldots, d_m) \cdot_{m,n} \mathrm{val}(d'_1, \ldots, d'_n), \tag{2}$$

$$\sum (d \mid d \in A) \cdot_{m,n} \sum (d' \mid d' \in B) = \sum (d \cdot_{m,n} d' \mid d \in A, d' \in B)\,. \tag{3}$$

Remark 3.2. Whenever the valuation monoid $(D, +, \mathrm{val}, \mathbb{0})$ is derived from a semiring $(D, +, \cdot, \mathbb{0}, \mathbb{1})$ where $\mathrm{val}(d_1, \ldots, d_m) = d_1 \cdot \ldots \cdot d_m$, then we can choose the products just as semiring multiplication, i.e., $\cdot_{m,n} := \cdot$ for all $m, n \in \mathbb{N}$. Then Equation (2) follows immediately and Equation (3) is just the distributivity of the semiring. Hence, all valuation monoids derived from semirings are Cauchy.

Definition 3.3. *The structure* $\big(D, +, \mathrm{val}, \mathrm{val}^\omega, (\cdot_{m,n} \mid m \in \mathbb{N}, n \in \mathbb{N} \cup \{\omega\}), \mathbb{0}\big)$ *is a* Cauchy ω-indexed valuation monoid *if* $\big(D, +, \mathrm{val}, (\cdot_{m,n} \mid m, n \in \mathbb{N}), \mathbb{0}\big)$ *is a Cauchy valuation monoid,* $\big(D, +, \mathrm{val}, \mathrm{val}^\omega, \mathbb{0}\big)$ *is an* ω*-indexed valuation monoid, and* $\cdot_{m,\omega} : D \times D \to D$ *for every* $m \in \mathbb{N}$ *such that for all* $d, d', d_i \in D$*, all finite subsets* $A \subseteq_{fin} D$ *and all subsets* $B \subseteq D$

$$\mathbb{0} \cdot_{m,\omega} d = d \cdot_{m,\omega} \mathbb{0} = \mathbb{0} \tag{4}$$

$$\mathrm{val}^\omega\big(n_i, d_i\big)_{i \geq 1} = d_1 \cdot_{n_1,\omega} \mathrm{val}^\omega\big(n_i, d_i\big)_{i \geq 2} \tag{5}$$

$$\sum (d \mid d \in A) \cdot_{m,\omega} \sum (d' \mid d' \in B) = \sum (d \cdot_{m,\omega} d' \mid d \in A, d' \in B), \tag{6}$$

and for every $C \subseteq_{fin} D$*,* $n_k \in \mathbb{N}$*, finite index sets* I_k*, and all* $d_{i_k} \in C$ $(i_k \in I_k)$

$$\mathrm{val}^\omega\Big(n_k, \sum_{i_k \in I_k} d_{i_k}\Big)_{k \in \mathbb{N}} = \sum_{(i_k)_k \in I_1 \times I_2 \times \ldots} \mathrm{val}^\omega\big(n_k, d_{i_k}\big)_{k \in \mathbb{N}}\,. \tag{7}$$

Definition 3.4. *A Cauchy* ω*-indexed valuation monoid* D *has the* partition property *if the following holds: For every* $(\boldsymbol{d_i})_{i \in \mathbb{N}} \in (D^+_{fin})^\omega = \bigcup_{C \subseteq_{fin} D} (C^+)^\omega$ *and every partition* $(I_j)_{j=1,\ldots,m}$ *of* $\mathbb{N}$ *into finitely many sets* $I_j = \{k^j_1, k^j_2, \ldots\}$ *with* $k^j_1 < k^j_2 < \ldots$*,*

we put $\boldsymbol{e_i^j} = \boldsymbol{d}_{\boldsymbol{k}_{i-1}^j+1} \dots \boldsymbol{d}_{\boldsymbol{k}_i^j}$ *for all* $i \in \mathbb{N}$ *(with* $k_0^j = 0$*), i.e.,* $(\boldsymbol{e_i^j})_{i\in\mathbb{N}}$ *is the same sequence of values from* D *as* $(\boldsymbol{d_i})_{i\in\mathbb{N}}$ *but in a coarser partition. Then*

$$\mathrm{val}^\omega\big(|\boldsymbol{d_i}|, \mathrm{val}(\boldsymbol{d_i})\big)_{i\in\mathbb{N}} = \sum_{\substack{j\in\{1,\dots,m\}\\ |I_j|=\omega}} \mathrm{val}^\omega\big(|\boldsymbol{e_i^j}|, \mathrm{val}(\boldsymbol{e_i^j})\big)_{i\in\mathbb{N}} \tag{8}$$

For the interpretation of these conditions, it is useful to consider val^ω as a parameterized (by ω-sequences over $\mathbb{N}$) infinitary product on D. Properties (2) and (5) are a kind of finitary associativity for val and val^ω. Distributivity of the parameterized products over sum is given by properties (3) and (6) whereas property (7) states distributivity of val^ω over finite sums. The partition property (8) is more subtle: in automata-theoretic terms it guarantees that we can compute the weight of an accepting run (with m acceptance states) as the sum of the weights of the same run but now with only one acceptance state (we choose one of the m accepting states), see Theorem 5.4(b). Under certain conditions property (8) is always satisfied:

Proposition 3.5. *Let* $(D, +, \mathrm{val}, \mathrm{val}^\omega, \mathbb{0})$ *be a Cauchy* ω*-indexed valuation monoid such that addition is idempotent, i.e.,* $d + d = d$ *for all* $d \in D$*. Let, moreover,* $\mathrm{val}^\omega(|\boldsymbol{d_i}|, \mathrm{val}(\boldsymbol{d_i}))_{i\in\mathbb{N}} = \mathrm{val}^\omega(|\boldsymbol{d'_i}|, \mathrm{val}(\boldsymbol{d'_i}))_{i\in\mathbb{N}}$ *for all* $(\boldsymbol{d_i})_{i\in\mathbb{N}}, (\boldsymbol{d'_i})_{i\in\mathbb{N}} \in (D^+_{fin})^\omega$ *with* $\boldsymbol{d_1 d_2} \dots = \boldsymbol{d'_1 d'_2} \dots \in D^\omega$ *(i.e., the concatenation of all finite sequences yields the same infinite sequence).*

Then $(D, +, \mathrm{val}, \mathrm{val}^\omega, \mathbb{0})$ *has the partition property (8).*

Remark 3.6. The complete star-omega-semirings of [12,13] or the semirings of [9] allow for infinite sums $\sum$ and products $\prod$ and fit into the setting of Cauchy ω-indexed valuation monoids. We define val^ω by $\mathrm{val}^\omega(n_i, d_i)_{i\in\mathbb{N}} = \prod_{i\in\mathbb{N}} d_i$. The parameterized products are just semiring multiplication. If the semiring is idempotent, then the associated Cauchy ω-indexed valuation monoid satisfies the conditions in Proposition 3.5. Hence, these structures have the partition property (8).

However, the kind of infinitary associativity described in Proposition 3.5 is not satisfied for ω-indexed valuation functions like limit superior (together with last) or limit average which we are especially interested in.

For notational simplicity, we will not always write down the products $\cdot_{m,n}$ as part of the structure when dealing with Cauchy valuation monoids $(D, +, \mathrm{val}, \mathbb{0})$. Next we give examples of Cauchy ω-indexed valuation monoids. Recall that we defined $\mathrm{last}(d, d') = d'$ if $d \neq \mathbb{0}$ and $\mathrm{last}(\mathbb{0}, d') = \mathbb{0}$.

Lemma 3.7. *The following* ω*-indexed valuation monoids are Cauchy* ω*-indexed valuation monoids satisfying the partition property (8):*

1. $(\overline{\mathbb{R}}, \sup, \mathrm{avg}, \limsup \mathrm{avg}, -\infty)$ *from Example 2.3 with the products*

$$d \cdot_{m,n} d' = \frac{m \cdot d + n \cdot d'}{m+n}, \quad d \cdot_{m,\omega} d' = \begin{cases} d' & \text{if } d \notin \{-\infty, \infty\} \text{ or } d' = -\infty, \\ d & \text{otherwise,} \end{cases}$$

2. $(\overline{\mathbb{R}}, \sup, \sup^{\mathbb{0}}, \limsup, -\infty)$ *from Example 2.4 with the products* $d \cdot_{m,n} d' = \sup^{\mathbb{0}}(d, d')$ *and* $d \cdot_{m,\omega} d' = \mathrm{last}(d, d')$*,*

3. $(\overline{\mathbb{R}}, \sup, \text{last}, \limsup, -\infty)$ *from Example 2.5 with the products* $d \cdot_{m,n} d' = d \cdot_{m,\omega} d' = \text{last}(d, d')$,
4. $(\overline{\mathbb{R}}_+, \sup, \text{disc}_\lambda, \limsup \text{disc}_\lambda, -\infty)$ *with* $0 < \lambda < 1$ *from Example 2.6 with the products* $d \cdot_{m,n} d' = d \cdot_{m,\omega} d' = d + \lambda^m d'$.

The ω-indexed valuation monoid with discounting was already explored in [8].

Now we are ready to define sums, concatenations, and iterations of series.

Definition 3.8. **(a)** *Let* $(D, +, \text{val}, \mathbb{0})$ *be a Cauchy valuation monoid and let* $S, S' : \Sigma^+ \to D$ *be two series. Then we define the* sum $S + S'$ *and the* Cauchy product $S \cdot S'$ *of* S *and* S' *by* $(S + S')(w) = S(w) + S'(w)$ *and*

$$(S \cdot S')(w) = \sum (S(u) \cdot_{|u|,|v|} S'(v) \mid u, v \in \Sigma^+, w = uv)$$

for every $w \in \Sigma^+$ *where we sum up over all factorizations of* w *into* $u, v \in \Sigma^+$. *We put* $S^1 = S$ *and* $S^{n+1} = S \cdot S^n$ *for all* $n \geq 1$.

The iteration S^+ *is defined by* $S^+(w) = \sum_{i=1}^{|w|} S^i(w)$.

(b) *Let* $(D, +, \text{val}, \text{val}^\omega, \mathbb{0})$ *be a Cauchy* ω*-indexed valuation monoid,* $S : \Sigma^+ \to D$, *and* $S', S'' : \Sigma^\omega \to D$. *The* sum $S' + S''$ *and the* Cauchy product $S \cdot S'$ *are defined for all* $w \in \Sigma^\omega$ *by* $(S' + S'')(w) = S'(w) + S''(w)$ *and*

$$(S \cdot S')(w) = \sum (S(u) \cdot_{|u|,\omega} S'(v) \mid u \in \Sigma^+, v \in \Sigma^\omega, w = uv).$$

The ω-iteration S^ω *of* $S : \Sigma^+ \to D$ *is defined for every* $w \in \Sigma^\omega$ *by*

$$S^\omega(w) = \sum \Big(\text{val}^\omega\big(|u_k|, S(u_k)\big)_{k \in \mathbb{N}} \mid w = u_1 u_2 \ldots \text{ for } u_k \in \Sigma^+ \Big)$$

where the sum is taken over all infinite factorizations $u_1 u_2 \ldots$ *of* w.

For semirings, the definitions of Cauchy products, iteration, and ω-iteration coincide with the classical definitions, cf. Remarks 3.2 and 3.6.

Proposition 3.9. **(a)** *Let* $(D, +, \text{val}, \mathbb{0})$ *be a Cauchy valuation monoid and* $S, S_1, S_2 : \Sigma^+ \to D$. *Then* $S \cdot (S_1 + S_2) = S \cdot S_1 + S \cdot S_2$ *and* $(S_1 + S_2) \cdot S = S_1 \cdot S + S_2 \cdot S$.

(b) *Let* $(D, +, \text{val}, \text{val}^\omega, \mathbb{0})$ *be a Cauchy* ω*-indexed valuation monoid,* $S : \Sigma^+ \to D$, *and* $S_1, S_2 : \Sigma^\omega \to D$. *Then* $S \cdot (S_1 + S_2) = S \cdot S_1 + S \cdot S_2$.

The class of *regular weighted expressions* over an alphabet Σ and a Cauchy valuation monoid $(D, +, \text{val}, \mathbb{0})$ is given by the grammar $E ::= d.a \mid E + E \mid E \cdot E \mid E^+$ where $d \in D$ and $a \in \Sigma$, cf. [21,17]. Let $d.a$ denote the series which maps a to d and all $w \in \Sigma^+ \setminus \{a\}$ to $\mathbb{0}$. The semantics of E is defined inductively by $[\![d.a]\!] = d.a$, $[\![E_1 + E_2]\!] = [\![E_1]\!] + [\![E_2]\!]$, $[\![E_1 \cdot E_2]\!] = [\![E_1]\!] \cdot [\![E_2]\!]$, and $[\![E^+]\!] = [\![E]\!]^+$.

The class of ω*-regular weighted expressions* over Σ and a Cauchy ω-indexed valuation monoid $(D, +, \text{val}, \text{val}^\omega, \mathbb{0})$ is given by the grammar $E ::= E + E \mid F \cdot E \mid F^\omega$ where F is any regular weighted expression. The semantics of E is the ω-series $[\![E]\!] : \Sigma^\omega \to D$ defined by $[\![E_1 + E_2]\!] = [\![E_1]\!] + [\![E_2]\!]$, $[\![F \cdot E]\!] = [\![F]\!] \cdot [\![E]\!]$, $[\![F^\omega]\!] = [\![F]\!]^\omega$.

Considering the limit average, cf. Example 2.3, $E = (1.a + -1.b)^\omega$ describes the long-run average difference between occurences of a and of b in some $w \in \{a, b\}^\omega$, e.g., $[\![E]\!](aaaababab \ldots) = 0$ and $[\![E]\!](abbabbabb \ldots) = -\frac{1}{3}$.

4 Weighted Finite Automata and Regular Expressions

Proposition 4.1. *Let $(D, +, \mathrm{val}, \mathbb{0})$ be a Cauchy valuation monoid. Let E and F be regular weighted expressions such that $[\![E]\!]$ and $[\![F]\!]$ are recognizable. Then $[\![E+F]\!]$, $[\![E \cdot F]\!]$, and $[\![E^+]\!]$ are recognizable.*

Proof idea. The disjoint union of the weighted automata recognizing $[\![E]\!]$ and $[\![F]\!]$, respectively, recognizes $[\![E + F]\!]$. Concerning $E \cdot F$, the respective automata $\mathcal{A}_E$ and $\mathcal{A}_F$ are concatenated consecutively where we introduce new intermediate states, given by pairs of final states from $\mathcal{A}_E$ and initial states of $\mathcal{A}_F$. Using conditions (2) and (3), we can show that the new automaton has the behavior $[\![E \cdot F]\!]$. Similarly, we proceed for E^+. Here, the weighted automaton $\mathcal{A}_E$ recognizing E is looped. □

Theorem 4.2. *Let $(D, +, \mathrm{val}, \mathbb{0})$ be a Cauchy valuation monoid and $S : \Sigma^+ \to D$. Then S is recognizable if and only if $S = [\![E]\!]$ for some regular weighted expression E.*

Proof sketch. Let $S = [\![E]\!]$. If $E = d.a$ with $d \in D$ and $a \in \Sigma$, then $[\![d.a]\!] = d.a$ is recognizable by the weighted automaton $(\{p, q\}, \{p\}, \{(p, a, q)\}, \{q\}, (p, a, q) \mapsto d)$. By Proposition 4.1 and induction on E, $S = [\![E]\!]$ is recognizable. Vice versa, let $\mathcal{A}$ be a weighted automaton. Then we build an expression E with $[\![E]\!] = \|\mathcal{A}\|$ by induction on the set of states used along a run. Here, we have to use the properties of Cauchy valuation monoids to ensure the correctness of the regular weighted expression constructed inductively. □

5 Weighted Büchi Automata and ω-Regular Expressions

We will show that for every ω-regular weighted expression E, the series $[\![E]\!]$ is ω-recognizable provided the underlying ω-indexed valuation monoid is Cauchy. If D satisfies additionally the partition property, then also the converse holds true.

By building the disjoint union of two Büchi automata, we can show

Proposition 5.1. *Let $(D, +, \mathrm{val}, \mathrm{val}^\omega, \mathbb{0})$ be an ω-indexed valuation monoid and let $S, S' : \Sigma^\omega \to D$ be ω-recognizable. Then $S + S'$ is ω-recognizable.*

Proposition 5.2. *Let $(D, +, \mathrm{val}, \mathrm{val}^\omega, \mathbb{0})$ be a Cauchy ω-indexed valuation monoid. Let $S : \Sigma^+ \to D$ be recognizable and $S' : \Sigma^\omega \to D$ be ω-recognizable. Then $S \cdot S' : \Sigma^\omega \to D$ is ω-recognizable.*

Proof sketch. Let $\mathcal{A} = (Q, I, T, F, \gamma)$ be a weighted automaton recognizing S and let $\mathcal{A}' = (Q', I', T', F', \gamma')$ be a weighted Büchi automaton recognizing S'. Let Q, Q', and $Q \times Q'$ be pairwise disjoint. We build a weighted Büchi automaton $\hat{\mathcal{A}} = (\hat{Q}, \hat{I}, \hat{T}, \hat{F}, \hat{\gamma})$ with $\hat{Q} = Q \cup Q' \cup (F \times I')$, $\hat{I} = I$, and $\hat{F} = (F \times I') \cup F'$. The transitions from $\hat{T}$ are those of T and T' together with copies $(p, a, (r, q))$ of $(p, a, r) \in T$ for $r \in F$ and copies $((r, q), a, p)$ of $(q, a, p) \in T'$ for $q \in I'$. The weights carry over, respectively. Here, not only the states from F' but also from $F \times I'$ are accepting, and a successful run passes a state from $F \times I'$ exactly once. This choice ensures the correct decomposition of every successful run $\hat{R}$ of $\hat{\mathcal{A}}$ by accepting states. Using this and properties (5) and (6), we can show that $\|\hat{\mathcal{A}}\| = S \cdot S'$. □

Proposition 5.3. *Let D be a Cauchy ω-indexed valuation monoid and $S : \Sigma^+ \to D$ be a recognizable series. Then S^ω is ω-recognizable.*

Proof idea. Let $S = \|\mathcal{A}\|$ for $\mathcal{A} = (Q, I, T, F, \gamma)$. We loop $\mathcal{A}$ by means of new intermediate states (r, q) with $r \in F$ and $q \in I$ which are as well the acceptance states of the constructed weighted Büchi automaton $\hat{\mathcal{A}}$. Now, any run $\hat{R} \in \mathrm{succ}(\hat{\mathcal{A}})$ can be decomposed by the acceptance states such that the emerging finite sub-runs of $\hat{R}$ are copies of successful runs from $\mathcal{A}$. By property (7), we conclude that $\|\hat{\mathcal{A}}\| = S^\omega$. □

Theorem 5.4. *Let D be a Cauchy ω-indexed valuation monoid.*

(a) *For any ω-regular weighted expression E, the ω-series $[\![E]\!]$ is ω-recognizable.*

(b) *Let D satisfy the partition property (8). Let $\mathcal{A}$ be a weighted Büchi automaton over D. Then there is an ω-regular weighted expression E with $[\![E]\!] = \|\mathcal{A}\|$.*

Proof sketch. **(a)** This follows by Theorem 4.2 and Propositions 5.1, 5.2, and 5.3.
(b) Constructing an equivalent ω-regular weighted expression from a weighted Büchi automaton is more involved than in the unweighted case. Let $\mathcal{A} = (Q, I, T, F, \gamma)$ be a weighted Büchi automaton. We define for $s, t \in Q$ the weighted Büchi automaton $\mathcal{A}^{st} = (Q, \{s\}, T, \{t\}, \gamma)$. First we show $\|\mathcal{A}\| = \sum_{s \in I, t \in F} \|\mathcal{A}^{st}\|$. Let $\mathrm{succ}_s(\mathcal{A})$ be the set of all successful runs of $\mathcal{A}$ starting in s. For $R \in \mathrm{succ}_s(\mathcal{A})$, we have by definition $\mathrm{wgt}_{\mathcal{A}}(R) = \mathrm{val}^\omega\big(|R_k|, \mathrm{val}\big(\gamma(R_k)\big)\big)_{k \in \mathbb{N}}$ where the R_k are the finite sub-runs of R running from one acceptance state to the next one (or from s to the first acceptance state if $k = 1$). We choose a partition $(I_t)_{t \in F}$ of $\mathbb{N}$ as follows: $I_t = \{k \mid R_k \text{ terminates in } t\}$. For $t \in F$, let $(R_k^t)_{k \in I_t}$ be the sequence of finite sub-runs of R such that R_1^t starts in s and ends in t and R_k^t for $k > 1$ starts and ends in t such that t does not appear in between. Now we conclude by the partition property (8) that $\|\mathcal{A}\| = \sum_{s \in I, t \in F} \|\mathcal{A}^{st}\|$.

Next we show that $\|\mathcal{A}^{st}\|$ can be described by an ω-regular expression. Using the analysis of runs in the proofs of Propositions 5.2 and 5.3 and properties (5) and (7), we can show that for suitable weighted finite automata $\mathcal{B}^{st}$ (to come from s to t) and $\mathcal{C}^t$ (to loop from t to t) we have $\|\mathcal{A}^{st}\| = \|\mathcal{B}^{st}\| \cdot \|\mathcal{C}^t\|^\omega$ whenever $s \neq t$ and $\|\mathcal{A}^{st}\| = \|\mathcal{C}^t\|^\omega$ if $s = t$. Due to Theorem 4.2 there are regular weighted expressions G^{st} and H^t with $[\![G^{st}]\!] = \|\mathcal{B}^{st}\|$ and $[\![H^t]\!] = \|\mathcal{C}^t\|$ and, thus, $\|\mathcal{A}\| = [\![E]\!]$ for the ω-regular weighted expression $E = \sum_{t \in I \cap F} (H^t)^\omega + \sum_{s \in I, t \in F, s \neq t} G^{st} \cdot (H^t)^\omega$. □

Remark 5.5. Our Theorem 5.4 generalizes previous results for discounting [8], for o-complete, infinitely distributive semirings [9], and for idempotent complete star-omega-semirings [12,13]. Ésik and Kuich give in [12,13] a general Kleene-like result even for non-idempotent semiring-semimodule pairs. However, this notion comprises an infinitary associativity for the product. Since we do not require this property and consider such valuation functions like limit superior or limit average, we have followed a combinatorial approach in our automaton model.

6 Conclusion

We have shown the equivalence of weighted automata and regular weighted expressions both for finite and infinite words in a new unified framework for the weighted structures. Our notion of Cauchy ω-indexed valuation monoid comprises such new weight

models like long-run peaks or long-run average. In another paper, we have developed also a characterization of weighted automata using valuation functions by fragments of weighted MSO logic. Thus, the main concepts to characterize the behavior of automata can be established also for these new weighted automata. Concerning expressions, the construction of small automata for a given expression is a vital topic. In this context, the method of partial derivatives was also transferred recently to the weighted setting [18]. Can we develop such methods also for this new kind of weighted automata?

References

1. Berstel, J., Reutenauer, C.: Rational Series and Their Languages. Springer, Heidelberg (1988)
2. Büchi, J.R.: On a decision method in restricted second order arithmetics. In: Proc. Intern. Congress on Logic, Methodology and Philosophy of Science, pp. 1–11. Stanford University Press, Stanford (1962)
3. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 385–400. Springer, Heidelberg (2008)
4. Chatterjee, K., Doyen, L., Henzinger, T.A.: Alternating weighted automata. In: Kutyłowski, M., Charatonik, W., Gębala, M. (eds.) FCT 2009. LNCS, vol. 5699, pp. 3–13. Springer, Heidelberg (2009)
5. Chatterjee, K., Doyen, L., Henzinger, T.A.: Expressiveness and closure properties for quantitative languages. In: 24th LICS 2009, pp. 199–208. IEEE Comp. Soc. Press, Los Alamitos (2009)
6. Chatterjee, K., Doyen, L., Henzinger, T.A.: Probabilistic weighted automata. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 244–258. Springer, Heidelberg (2009)
7. Droste, M., Kuich, W., Vogler, H. (eds.): Handbook of Weighted Automata. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (2009)
8. Droste, M., Kuske, D.: Skew and infinitary formal power series. Theoretical Computer Science 366, 199–227 (2006)
9. Droste, M., Püschmann, U.: Weighted Büchi automata with order-complete weights. Int. J. of Algebra and Computation 17(2), 235–260 (2007)
10. Droste, M., Vogler, H.: Kleene and Büchi theorems for weighted automata and multi-valued logics over arbitrary bounded lattices. In: Gao, Y., Lu, H., Seki, S., Yu, S. (eds.) DLT 2010. LNCS, vol. 6224, pp. 160–172. Springer, Heidelberg (2010)
11. Eilenberg, S.: Automata, Languages, and Machines, vol. A. Academic Press, London (1974)
12. Ésik, Z., Kuich, W.: A semiring-semimodule generalization of ω-regular languages I+II. Journal of Automata, Languages and Combinatorics 10, 203–242 & 243–264 (2005)
13. Ésik, Z., Kuich, W.: Finite automata. In: Droste, et al. (eds.) [7], ch. 3
14. Kleene, S.: Representations of events in nerve nets and finite automata. In: Shannon, C., McCarthy, J. (eds.) Automata Studies, pp. 3–42. Princeton University Press, Princeton (1956)
15. Kuich, W.: Semirings and formal power series: their relevance to formal languages and automata. In: New Trends in Formal Languages, vol. 1, ch.9. Springer, Heidelberg (1997)
16. Kuich, W., Salomaa, A.: Semirings, Automata, Languages. EATCS Monographs in Theoretical Computer Science. Springer, Heidelberg (1986)
17. Kupferman, O., Lustig, Y.: Lattice automata. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 199–213. Springer, Heidelberg (2007)

18. Lombardy, S., Sakarovitch, J.: Derivatives of rational expressions with multiplicity. Theoretical Computer Science 332, 141–177 (2005)
19. Rahonis, G.: Infinite fuzzy computations. Fuzzy Sets and Systems 153, 275–288 (2005)
20. Salomaa, A., Soittola, M.: Automata-Theoretic Aspects of Formal Power Series. Texts and Monographs in Computer Science. Springer, Heidelberg (1978)
21. Schützenberger, M.: On the definition of a family of automata. Information and Control 4, 245–270 (1961)

Reachability Games on Automatic Graphs

Daniel Neider

Lehrstuhl für Informatik 7, RWTH Aachen University, Germany

Abstract. In this work we study two-person reachability games on finite and infinite automatic graphs. For the finite case we empirically show that automatic game encodings are competitive to well-known symbolic techniques such as BDDs, SAT and QBF formulas. For the infinite case we present a novel algorithm utilizing algorithmic learning techniques, which allows to solve huge classes of automatic reachability games.

1 Introduction

Infinite games on graphs are a general modeling tool in the analysis and synthesis of (reactive) systems. Their roots go back to Church's synthesis problem where he described the task of automatically synthesizing circuits from given specifications. Since then much research has been done, especially in using games for model checking and formal verification (e.g. for the μ-calculus [1]).

Surprisingly, algorithmic learning has not been considered in the original context of infinite games so far, although it has been successfully applied to model checking (e.g. in [2] and [3]). The purpose of this paper is to study where known algorithms can benefit from learning techniques as a first step towards filling this gap.

In order to apply learning techniques for regular languages, games have to be encoded by means of finite automata in an appropriate way. For this purpose we introduce *automatic games*, i.e. infinite games that are played on automatic graphs. Thereby, an automatic graph is a directed (finite or infinite) graph, whose vertices form a regular set and whose edges are an automatic relation accepted by a finite state transducer (cf. [4]).

In this paper we focus on two player *reachability games* since this type of games is the most fundamental one. The objective is to compute a winning strategy for the first player that forces each play to eventually visit a "good" state no matter how his opponent plays. Reachability games typically occur when verifying guarantee and safety properties, but most solutions for more complex games computationally rely on them.

The first part of this work is dedicated to reachability games on *finite graphs*. We present an algorithm, based on computing fixed points (but not yet using learning) for such games. A framework introduced in [5] makes it possible to compare this approach to other symbolic techniques such as binary decision diagrams (BDDs) and transformations into propositional logic (SAT) and quantified Boolean formulas (QBF). We implemented our algorithm and the results are very promising showing that automatic encodings are competitive to the methods benchmarked in [5].

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 222–230, 2011.

In the second part we consider reachability games on *infinite graphs*. In this scenario fixed-point computations do not necessarily converge within finitely many steps. In the context of model checking various acceleration techniques have been developed an applied successfully to overcome this problem. Perhaps the most important example in this context is *widening* (used e.g. in [6]).

However, we present a novel, complementary approach adapted from [3], which applies techniques known from algorithmic learning. The idea is to learn a fixed point (provided that it can be represented as a regular set) instead of computing it iteratively. This has the advantage that termination and the runtime does not depend on how long it takes to compute the fixed point, and, hence, this procedure also works when the fixed point does not converge within a finite bound. Moreover, since our algorithm also works on finite game graphs, this procedure can be applied to a wide range of automatic reachability games.

This paper is organized as follows. In Section 2 we introduce our notation of infinite games and define automatic games. In Section 3 we describe a fixed-point-based algorithm to solve automatic reachability games on finite graphs and compare it with other symbolic techniques. Next, in Section 4 we present our learning based algorithm that solves automatic reachability games on infinite graphs. Finally, Section 5 concludes and presents future work.

2 Preliminaries

An *infinite game* $\mathcal{G} = (\mathcal{A}, \mathrm{Win})$ is composed of an *arena* $\mathcal{A} = (V_0, V_1, E)$ and a *winning condition* $\mathrm{Win} \subseteq (V_0 \cup V_1)^\omega$. The underlying arena is a graph with two disjoint sets of vertices V_0, V_1—the set of all vertices is denoted by $V = V_0 \cup V_1$—and an edge-relation $E \subseteq V \times V$. We assume that each vertex has at least one and at most finitely many successors. The first requirement is to ease the notation only and, therefore, no restriction. The latter requirement, however, restricts the considered class of graphs, but is necessary to guarantee the termination of our learning algorithm.

A game is played by two players, *player 0* and *player 1*, who move a token along the edges forming a play. Formally, a *play* $\pi = v_0 v_1 \ldots \subseteq V^\omega$ is an infinite sequence of vertices that respects the edge relation. Player 0 wins a play if $\pi \in \mathrm{Win}$; then, the play is called *winning for player 0.* Conversely, a play is winning for player 1 if $\pi \notin \mathrm{Win}$.

In the analysis of games one is interested in strategies $f\colon V^*V_\sigma \to V$ for a player $\sigma \in \{0, 1\}$ that maps a play prefix $v_0 v_1 \ldots v_n$ to a successor vertex v_{n+1} with $(v_n, v_{n+1}) \in E$. A strategy is *winning* for player σ from vertex v if every play starting in v and conforming with f is winning for him. The *winning region* W_σ is the set of vertices from which player σ has a winning strategy. The term *solving a game* here refers to computing winning strategies and winning regions.

In a *reachability game* the winning condition is given as a set $F \subseteq V$ and a play is winning for player 0 if it eventually reaches a vertex in F; otherwise it is winning for player 1. As an abbreviation we write $\mathcal{G} = (A, F)$ when referring to reachability games.

To encode infinite graphs symbolically, regular sets are a popular means, e.g. as in regular model checking [7]. The idea is to label each vertex uniquely with a finite string over some alphabet Σ and to represent sets of vertices and edges as regular sets. The resulting graphs are called *automatic* (cf. [4]).

Formally, an *automatic game* $\mathcal{G} = (\mathcal{A}, \mathrm{Win})$ is an infinite game played on an automatic graph. More precisely, the sets V_0, V_1 are regular sets of vertex labels and the edge relation E is an automatic relation given as a finite state transducer τ. This transducer reads pairs of vertex labels synchronously letter-by-letter and accepts if there is an edge between them. Thereby, a special $\bot$-symbol not contained in Σ is used to equalize the length of two labels if the label of one vertex is longer then the other (i.e. τ works over the alphabet $(\Sigma \cup \{\bot\})^2$). Figure 1 and 2 show an example of an automatic game arena.

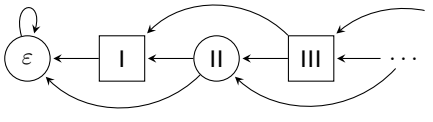


Fig. 1. A simple game arena. Player 0 vertices are depicted as circles, player 1 vertices as squares. The alphabet used to label the vertices is $\Sigma = \{\mathrm{I}\}$.

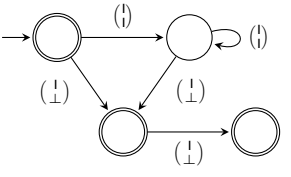


Fig. 2. A finite state transducer encoding the edge-relation of the arena shown in Figure 1.

Additionally, the winning condition must also be represented by means of finite automata. In the case of reachability games we simply require that $F \subseteq V$ is a regular set of vertex labels.

For the rest of this paper we do no longer distinguish between a vertex and its label since we require a one-to-one relation; simply think of the set of all vertices as a language over some alphabet Σ, i.e. $V \subseteq \Sigma^*$. Moreover, whenever we refer to regular sets we assume that they are represented as finite automata.

3 Automatic Reachability Games on Finite Arenas

In this section we concentrate on reachability games where the arena is finite, i.e. $|V|$ is finite. Madhusudan, Nam and Alur [5] provided a common framework to compare different symbolic techniques for this scenario and already benchmarked algorithms relying on BDD representations as well as SAT and QBF formulas. In the following we show how we solve automatic reachability games on finite arenas, then introduce our proof-of-concept implementation, and compare our results to the findings in [5].

Solving Automatic Reachability Games on Finite Arenas. To solve an automatic reachability game $(\mathcal{A}, F)$, we use the standard fixed-point algorithm to compute attractor sets: for $X \subseteq V$ and $i \in \mathbb{N}$ let $\mathrm{Attr}^0_\sigma(X) = X$ and

$$\mathrm{Attr}^i_\sigma(X) = \mathrm{Attr}^{i-1}_\sigma(X) \cup \{v' \in V_\sigma \mid \exists v \in V\colon\ (v', v) \in E \ \wedge\ v \in \mathrm{Attr}^{i-1}_\sigma(X)\} \cup \{v' \in V_{1-\sigma} \mid \forall v \in V\colon\ (v', v) \in E \ \rightarrow\ v \in \mathrm{Attr}^{i-1}_\sigma(X)\}.$$

The σ-*attractor* of X, i.e. the set of vertices from which player σ can force a visit of a vertex in X, is the infinite union $\mathrm{Attr}_\sigma(X) = \bigcup_{i\in\mathbb{N}} \mathrm{Attr}^i_\sigma(X)$.

The winning region W_0 (respectively W_1) is then given by the set $\mathrm{Attr}_0(F)$ (respectively $W_1 = V \setminus W_0$). Moreover, using the attractor of player σ it is easy to extract a corresponding winning strategy: from a vertex $v \in \mathrm{Attr}^i_\sigma(F) \cap V_\sigma$ for all $i \geq 0$ move to a vertex v' such that $(v, v') \in E$ and $v' \in \mathrm{Attr}^{i-1}_\sigma(F)$.

We compute the attractor of a regular set X of vertices iteratively for increasing values of i. Each such step can be performed symbolically: to compute all vertices of player σ belonging to $\mathrm{Attr}^{i+1}_\sigma(X)$, we first compute all predecessors of vertices in $\mathrm{Attr}^i_\sigma(X)$. Thereto, we apply cylindrification to the set $\mathrm{Attr}^i_\sigma(X)$, i.e. we extend the alphabet to match the one of the transducer τ, obtaining the set $(\Sigma \cup \{\bot\})^* \times \mathrm{Attr}^i_\sigma(X)$ and compute the intersection with τ. Then, we project the result on its first component and intersect it with the vertices of player σ. To compute the corresponding vertices of player $1 - \sigma$, we repeat the same operations for the set $V \setminus \mathrm{Attr}^i_\sigma(X)$ and complement the result with respect to V. Finally, we compute the union of both sets and $\mathrm{Attr}^i_\sigma(X)$.

The exact construction is a bit tedious and can be done using standard automata operations including Boolean operations, projection and determinization. As result we obtain that for every (regular) set $X \subseteq V$ the set $\mathrm{Attr}^i_\sigma(X)$ is also regular for all $i \in \mathbb{N}$. Moreover, since we deal with finite arenas, the attractor computation becomes stationary after at most $|V|$ steps. This means that $\mathrm{Attr}_\sigma(F) = \mathrm{Attr}^{|V|}_\sigma(F)$ is a regular set and can be computed as described above.

Experiments and Results. The framework presented in [5] covers two reachability games: the peruser-evader game and the swap game. Originally, these games involve concurrent interaction between both players (so-called *concurrent reachability games*) and the players typically play randomized strategies, which win with a certain probability. However, if one is interested in strategies that win every play (not only with probability 1), then one can easily adapt the games to match our setting (cf. [8]).

We implemented the symbolic algorithm described above in the Java programming language using the *dk.brics.automaton* automaton library. All experiments were done on a PC using a 2.5 GHz Pentium E5200 processor (only one core was used), 1.5 GB RAM, and running Linux. Note that our experimental setting is different from [5] w.r.t. to computational power, but we are more interested in the size of the instances that can be handled than in the runtime.

We encoded each game in three different ways. For the first two encodings we choose a unary (respective binary) representation to model these games. For the third encoding we used the symbols of the alphabet itself to model positions in a game. This results in much smaller arenas but requires a bigger alphabet. Our goal was to see whether there is a trade-off between the number of states necessary to encode an automatic game and the alphabet size. Note that the alphabet size $|\Sigma|$ of the first two encodings is independent of the size of the game arena while it grows in the latter encoding.

Tables 1 and 2 show the results of our implementation for the pursuer-evader and the swap game. The column labeled with "Steps" displays the number of

iterations until the the attractor computation became stationary. (Note that the number of steps depends on how the game is modeled as a game graph.) The figures in the "States" column show how many states the resulting automaton for the attractor contains. A "–" indicates either that Java ran out of memory or that the computation did not finish within 10 hours (which is the limit in [5]).

Table 1. Results pursuer-evader game

Encoding	Grid size	Time (in sec.)	Steps	States	$\lvert\Sigma\rvert$
Unary	16×16	284	46	18286	3
	32×32	25551	94	131486	3
	64×64	–	–	–	–
Binary	16×16	29	46	3895	3
	32×32	663	94	17624	3
	64×64	32613	190	75283	3
Alphabet	16×16	24	46	1004	17
	32×32	601	94	3868	33
	64×64	–	–	–	–

Table 2. Results swap game

Encoding	Grid size	Time (in sec.)	Steps	States	$\lvert\Sigma\rvert$
Unary	6	612	8	1173	3
	7	11139	10	2881	3
	8	–	–	–	–
Binary	6	405	8	1077	3
	7	2740	10	1288	3
	8	–	–	–	–
Alphabet	6	11	8	126	8
	7	464	10	254	9
	8	10275	12	510	10

In the case of the pursuer-evader game we were able to solve games up to a grid of size 32×32 (for binary encoding up to 64×64). On the other hand, the swap game could be solved for arrays of size 7 (respectively 8 when using the alphabet encoding). However, using different ways to encode games seems to have no significant influence on the viable size of the game arenas.

The benchmark in [5] shows similar results. BDD methods were able to solve the pursuer-evader game up to a grid-size of 32×32 (only the Mucke tool performed significantly better up to 512×512) and arrays up to a length of 9 for the swap game. The SAT and BQF encodings, on the other hand, could not match the result of our implementation. Both methods could do at most 15 steps in the pursuer-evader game and 7 steps in the swap game (depending on the size of the arena). Both figures, however, are significantly worse than our results.

As the main result we obtain that automatic encodings are competitive to the symbolic methods studied in [5]. However, let us emphasize that our implementation is just a prototype and not as optimized as the tools used in [5].

4 Automatic Reachability Games on Infinite Arenas

In this section we consider automatic reachability games on infinite game arenas, which typically occur when analyzing systems that have access to auxiliary memory. Unfortunately, fixed-point computations, as used in Section 3, do no longer guarantee to converge and terminate in finite time. To overcome this problem, we introduce a learning based algorithm that actively learns the desired fixed point in interaction with a teacher.

It seems that there is no satisfactory way to learn the attractor directly. Therefore, we introduce an alternative characterization of the attractor, using a special functional Γ_σ, that can be learned. The main task here is to construct a

teacher that can answer the queries asked by a learning algorithm. Finally, we plug in our preferred learning algorithm and compute the attractor.

Learning from a Minimally Adequate Teacher. A suitable setting for our purpose was introduced by Angluin in [9]. There, a learner learns a regular language $L \subseteq \Sigma^*$ over an a priory fixed alphabet Σ from a *minimally adequate teacher*. This teacher is capable of answering *membership* and *equivalence queries*. On a membership query the teacher is provided with a word $w \in \Sigma^*$ and has to answer whether $w \in L$. On an equivalence query a hypothesis is given, typically as an automaton $\mathcal{A}$, and the teacher has to check whether $\mathcal{A}$ is an equivalent description of L. If so, he returns "yes". Otherwise, the teacher is required to return a counter-example $w \in L(\mathcal{A}) \triangle L = (L \setminus L(\mathcal{A})) \cup (L(\mathcal{A}) \setminus L)$, i.e. a witness that L and $L(\mathcal{A})$ are different.

In [9] Angluin showed that for every regular language L the smallest deterministic automaton accepting L can be learned from a minimally adequate teacher in polynomial time. In the following, it is therefore enough to construct a teacher that can answer membership and equivalence queries regarding the fixed point we want to learn.

Fixed-Point Characterization of the Attractor. Learning the set $\mathrm{Attr}_\sigma(F)$ itself is problematic. Already answering membership queries "$w \in \mathrm{Attr}_\sigma(F)$?" is intricate and there seems to be no satisfactory way to do so. Thus, the main idea, adapted from [3], is to add additional information to the fixed point: instead of learning $\mathrm{Attr}_\sigma(F)$, we learn a set of pairs (v, i) where $v \in V$ is a vertex and $i \in \mathbb{N}$ is a natural number. The meaning of such a pair is that player σ can force to visit a vertex in F in at most i moves, i.e. $v \in \mathrm{Attr}^i_\sigma(F)$.

Formally, let $\mathcal{G} = (\mathcal{A}, F)$ be an automatic reachability game and $X \subseteq V \times \mathbb{N}$ be a set containing pairs of the form (v, i) where $v \in V$ is a vertex and $i \in \mathbb{N}$ is a natural number. Moreover, let $\sigma \in \{0, 1\}$ denote either player 0 or player 1. We define a functional $\Gamma_\sigma \colon 2^{V \times \mathbb{N}} \to 2^{V \times \mathbb{N}}$ (depending on the game $\mathcal{G}$) such that $\Gamma_\sigma(X) = F \times \mathbb{N} \cup \gamma_\sigma(X) \cup \gamma_{1-\sigma}(X)$ where

$$\gamma_\sigma(X) = \big\{(v, i+1) \mid v \in V_\sigma \text{ and } \exists v' \in V \text{ with } (v, v') \in E \colon (v', i) \in X\big\},$$
$$\gamma_{1-\sigma}(X) = \big\{(v, i+1) \mid v \in V_{1-\sigma} \text{ and } \forall v' \in V \text{ with } (v, v') \in E \colon (v', i) \in X\big\}.$$

Intuitively, Γ_σ "simulates" one step in the attractor computation of player σ for an arbitrary set X taking the distance information into account. Thereby, the computation of Γ_σ can be done symbolically similar to the construction in Section 3. This means that if X is a regular set, then $\Gamma_\sigma(X)$ is also regular.

Note that Γ_σ is monotone and, hence, has a fixed point. As Lemma 1 shows, a fixed point of Γ_σ is a slightly different but complete characterization of $\mathrm{Attr}_\sigma(F)$ and, thus, unique. This fixed point is what we are going to learn. The lemma itself is proved by induction over i.

Lemma 1. *If X is a fixed point of Γ_σ, then $(v, i) \in X \Leftrightarrow v \in \mathrm{Attr}^i_\sigma(F)$ holds.*

A Fixed Point Teacher. Next, we describe how a teacher capable of answering membership and equivalence queries regarding the (unique) fixed point X of Γ_σ works. Unfortunately, it is not guaranteed that this fixed point actually is a regular set since regular languages are not closed under infinite union. We discuss this restriction later on and assume here that the fixed point is regular.

Answering Membership Queries. Answering membership queries "$(v, i) \in X$?" is easy: perform a forward-search for i steps starting at the vertex v and check whether player σ can force to visit a vertex $v' \in F$.

Answering Equivalence Queries. On an equivalence query, we are given a regular set Y and have to check whether Y is the desired fixed point X of Γ_σ. For this we compute $\Gamma_\sigma(Y)$. Since Γ_σ has a unique fixed point, we can return "yes" if $Y = \Gamma_\sigma(Y)$ holds. In any other case, we have to compute a counter-example in the symmetric difference $X \triangle Y$, which is done as follows.

1. case. Let $\Gamma_\sigma(Y) \setminus Y \neq \emptyset$ and $(v, i) \in \Gamma_\sigma(Y) \setminus Y$. If $i = 0$, then we know by definition of Γ_σ that $v \in F$ and, hence, $(v, i) \in X$. Thus, $(v, i) \in X \triangle Y$. If $i > 0$, we ask a membership query on (v, i). If $(v, i) \in X$, then clearly $(v, i) \in X \triangle Y$. Otherwise we distinguish whether $v \in V_\sigma$ or $v \in V_{1-\sigma}$. In the first case, for every $v' \in V$ with $(v, v') \in E$ and $(v', i-1) \in Y$ we know that $(v', i-1) \notin X$ (otherwise $(v, i) \in X$). Thus, $(v', i-1) \in X \triangle Y$. In the second case, there is at least one $v' \in V$ with $(v, v') \in E$ and $(v', i-1) \in Y$ such that $(v', i-1) \notin X$. We find such a $(v', i-1)$ by asking membership queries. Note that this procedure terminates since we required every node to have only finitely many outgoing edges.

2. case. Let $\Gamma_\sigma(Y) \subsetneq Y$. Then, Y is a so-called *pre-fixed point*. From fixed-point theory we know that the intersection of all pre-fixed points yields again a fixed point, which is in this case the set X. Since Γ_σ is monotone, we deduce that $\Gamma_\sigma(\Gamma_\sigma(Y)) \subseteq \Gamma_\sigma(Y)$ and, hence, $\Gamma_\sigma(Y)$ is also a pre-fixed point. This means that every element $(v, i) \in Y \setminus \Gamma_\sigma(Y)$ is not in the intersection of all pre-fixed points and, therefore, $(v, i) \notin X$, i.e. $(v, i) \in X \triangle Y$.

Note that answering membership and equivalence queries can be done symbolically for automatic reachability games using standard automata operations.

Solving Automatic Reachability Games on Infinite Arenas. Computing the winning regions and the winning strategy in automatic reachability games on finite and infinite arenas is done as follows. We construct a teacher for the fixed point X of Γ_σ and plug in our preferred learning algorithm (e.g. Angluin's L^* algorithm [9]). After learning X, we obtain $\mathrm{Attr}_\sigma(F)$ by projecting X on the first component. The attractor can now be used to extract the winning regions and the winning strategy for player σ. Altogether, we obtain our main result.

Theorem 1. *Let $\mathcal{G}$ be an automatic reachability game. Then, the winning regions W_0, W_1 and winning strategies for both players can be computed by learning the fixed point of Γ_σ if the fixed point can be represented as a regular set.*

Unfortunately, the decision problem "Given an automatic reachability game. Is the fixed point of Γ_σ regular?" is undecidable. This result is due to the fact that it is possible to encode computations of Turing machines in automatic graphs.

The time and space complexity of our approach depends on the number of queries the chosen learning algorithm asks (this number is polynomial when using the L* algorithm). Answering an equivalence query requires to construct at most d automata of size $2^{\mathcal{O}(n \cdot t)}$ where n is the number of states of the fixed point X, t the number of states of the transducer τ and d is a bound for the number of outgoing edges per vertex. Moreover, the length m of a counter-example can also be bound by $2^{\mathcal{O}(n \cdot t)}$, and during an equivalence query at most $d + 1$ membership queries are asked. Membership queries can be answered by a forward-search in the game graph, which requires $d^{2^{\mathcal{O}(n \cdot t)}}$ steps. Altogether we obtain a doubly-exponential algorithm in the size of the automatic game.

An extension of our proof-of-concept from Section 3 using the libalf learning framework [10] shows good results on many infinite arenas. A speed-up for the finite case (on the examples from Section 3), however, could not be observed.

5 Conclusion

The contributions of this paper are twofold. First, we showed using a proof-of-concept implementation of the fixed-point algorithm that automatic encodings are a competitive means for solving reachability games on finite arenas. Second, we introduced a learning based algorithm suitable to solve automatic reachability games. An extension of our proof-of-concept proves that the algorithm works for many infinite arenas. To the best of our knowledge this work is the first in which learning techniques have been applied to solve infinite games.

The latter result is of special interest as it offers a general framework. First, it allows to solve reachability games for every suitable symbolic representation in which sets of pairs (v, i) can be represented, Γ_σ can be computed and a learning algorithm is available. Second, adding further information to the fixed point allows to solve more complex games, e.g. games with Büchi winning conditions (cf. [3]). For further research we want to extend this work to more expressive representations, e.g. (visibly) pushdown automata, and more general winning conditions such as parity or Muller conditions.

References

1. Thomas, W.: Infinite games and verification. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 58–64. Springer, Heidelberg (2002)
2. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. Electr. Notes Theor. Comput. Sci. 138(3), 21–36 (2005)
3. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Using language inference to verify omega-regular properties. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 45–60. Springer, Heidelberg (2005)
4. Blumensath, A., Grädel, E.: Finite presentations of infinite structures: Automata and interpretations. Theory Comput. Syst. 37(6), 641–674 (2004)
5. Alur, R., Madhusudan, P., Nam, W.: Symbolic computational techniques for solving games. STTT 7(2), 118–128 (2005)

6. Touili, T.: Regular model checking using widening techniques. Electr. Notes Theor. Comput. Sci. 50(4) (2001)
7. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
8. de Alfaro, L., Henzinger, T.A., Kupferman, O.: Concurrent reachability games. Theor. Comput. Sci. 386(3), 188–217 (2007)
9. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
10. Bollig, B., Katoen, J.P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: The Automata Learning Framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010)

Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions*

Satoshi Okui[1] and Taro Suzuki[2]

[1] Department of Computer Science, Chubu University, Japan
okui@cs.chubu.ac.jp
[2] School of Computer Science and Engineering, The University of Aizu, Japan
taro@u-aizu.ac.jp

Abstract. This paper offers a new efficient regular expression matching algorithm which follows the POSIX-type leftmost-longest rule. The algorithm basically emulates the subset construction without backtracking, so that its computational cost even in the worst case does not explode exponentially; the time complexity of the algorithm is $O(mn(n+c))$, where m is the length of a given input string, n the number of occurrences of the most frequently used letter in a given regular expression and c the number of subexpressions to be used for capturing substrings. A formalization of the leftmost-longest semantics by using parse trees is also discussed.

1 Introduction

Disambiguation in regular expression matching is crucial for many applications such as string replacement in document processing. POSIX 1003.2 standard requires that the ambiguity of regular expressions to be resolved by following the *leftmost-longest* rule.

For implementations that rely on backtrack, strictly following the leftmost-longest rule forces loss of efficiency; the greedy, or the first match, policy [6] is easier for such systems to follow. For this problem, some recent implementations, e.g., [9,8], take approaches based on automata, realising the leftmost-longest semantics efficiently.

This paper presents a new regular expression matching algorithm which follows the leftmost-longest semantics. The time complexity of our algorithm is $O(mn(n+c))$ where m is the length of a given input string, n the number of occurrences of the most frequently used letter in a given regular expression and c the number of subexpressions to be used for *capturing* portions of the input string.

Our discussion for achieving that algorithm proceeds in three steps as follows. First, in Section 2, we formalize the leftmost-longest rule based on *parse trees*. A parse tree for a given input string represents a way of accepting that string; e.g., which alternative is selected or how many times each iteration is performed, and so forth. Imposing a priority order on parse trees results in a straightforward interpretation of the leftmost-longest rule, which serves as a basis for our later discussion. We restrict ourselves to consider *canonical* parse trees for avoiding the matches which is unacceptable in the

* This work is supported by Japan Society for Promotion of Science, Basic Research (C) No.22500019.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 231–240, 2011.

leftmost-longest semantics. Next, Section 3 introduces a slight extension of traditional *position automata* [10,7] in order to enumerate canonical parse trees. In Section 4, we introduce the regular expression matching algorithm, which basically performs subset construction at runtime. This algorithm incrementally compares, in each step, the priority of any two paths to eliminate unnecessary ones. Finally, Section 5 is devoted to a brief comparison with other studies.

Due to limitations of space, correctness issues of our algorithm are not discussed here. These are found in an extended version of this paper (`http://www.scl.cs.chubu.ac.jp/reg2010/`).

2 Formalizing the Leftmost-Longest Semantics

A *regular expression* is an expression of the form: $r ::= 1 \mid a \mid r{\cdot}r \mid r{+}r \mid r*$ where 1 is the null string, a a letter in the alphabet Σ and $r_1{\cdot}r_2$ the concatenation of r_1 and r_2. Our definition does not include the regular expression for the empty language. We think of the regular expressions as abstract syntax trees (AST). For strings accepted by a regular expression r, we consider the set $\mathtt{PT}(r)$ of *parse trees*:

$$\mathtt{PT}(1) = \{1\} \quad \mathtt{PT}(a) = \{a\} \quad \mathtt{PT}(r_1{\cdot}r_2) = \{t_1{\cdot}t_2 \mid t_i \in \mathtt{PT}(r_i), i = 1, 2\}$$
$$\mathtt{PT}(r_1{+}r_2) = \{\mathtt{L}(t) \mid t \in \mathtt{PT}(r_1)\} \cup \{\mathtt{R}(t) \mid t \in \mathtt{PT}(r_2)\}$$
$$\mathtt{PT}(r*) = \{\mathtt{I}(t_1, \ldots, t_n) \mid t_1, \ldots, t_n \in \mathtt{PT}(r), n \geq 0\}$$

Note that parse trees are *unranked* in the sense that an `I`-node has an arbitrary number of children (possibly none). Parse trees for strings should not be confused with the AST of the regular expression; basically, parse trees are obtained from the AST of a regular expression by horizontally expanding children of $*$-nodes and choosing the left or right alternatives for +-nodes.

A *position* (also called a *path string*) is a sequence of natural numbers. The root position is the empty string and denoted as Λ. The child of the `R`-node has a position ending with 2 rather than 1. For other kind of nodes, the first sibling always has a position ending with 1, the next sibling has a position ending with 2 and so forth. The length of a position p (as string) is denoted by $|p|$. The subterm of t at position p is denoted as $t|_p$. We write $p_1 \lessdot p_2$ iff p_1 precedes p_2 in lexicographic order.

Let t be a parse tree. The *norm* of t at $p \in \mathtt{Pos}(t)$, written $\| t \|_p$, is the number of letters (of Σ) in $t|_p$ (Note: we do not count 1-nodes). For p not in $\mathtt{Pos}(t)$, we define $\| t \|_p = -1$. The subscript of $\| t \|_\Lambda$ is omitted.

According to the POSIX specification [1], "*a subexpression repeated by '*' shall not match a null expression unless this is the only match for the repetition.*" For example, for a regular expression, $a{**}$, $\mathtt{I}(\mathtt{I}())$ satisfies this requirement while $\mathtt{I}(\mathtt{I}(), \mathtt{I}())$, $\mathtt{I}(\mathtt{I}(), a)$ or $\mathtt{I}(a, \mathtt{I}())$ does not. This leads us to the notion of *canonical* parse tree:

Definition 1. *(canonical parse tree) A parse tree t is called* canonical *if any subterm of t such as $\mathtt{I}(t_1, \ldots, t_n)$, $n \geq 2$, satisfies $\| t_i \| > 0$ for any $1 \leq i \leq n$.*

Obviously, a leaf of a parse tree is either 1 or a letter in Σ. Reading those letters in Σ from left to right, we obtain the string *derived* from that parse tree. We write $\mathtt{CPT}(r, w)$ for the set of canonical parse trees deriving w.

The specification [1] also describes that "*consistent with the whole match being the longest of the leftmost matches, each subpattern, from left to right, shall match the longest possible string*." This leads the following definition of priority:

Definition 2. *For any* $t_1, t_2 \in \mathtt{CPT}(r)$*, we say* t_1 is prior to t_2*, written* $t_1 \lessdot t_2$*, if the following conditions are satisfied for some* $p \in \mathtt{Pos}(t_1) \cup \mathtt{Pos}(t_2)$*:*

1. $\| t_1 \|_p > \| t_2 \|_p$
2. $\| t_1 \|_q = \| t_2 \|_q$ *for any position* $q \in \mathtt{Pos}(t_1) \cup \mathtt{Pos}(t_2)$ *such that* $q \lessdot p$.

Recall that we define $\| t \|_p = -1$ for $p \notin \mathtt{Pos}(t)$. That corresponds to the requirement [1]: "*a null string shall be considered to be longer than no match at all*." The specification also states that "*the search for a matching sequence starts at the beginning of a string and stops when the first sequence matching the expression is found*" and that "*if the pattern permits a variable number of matching characters and thus there is more than one such sequence starting at that point, the longest such sequence is matched*," which we formalize as follows.

Given a regular expression r and a string w, we define the set $\mathtt{PC}(r, w)$ of *parse configurations* for r and w as $\{\langle u, t, v\rangle \mid t \in \mathtt{CPT}(r, w'), uw'v = w\}$. For $\langle u_1, t_1, v_1\rangle$ and $\langle u_2, t_2, v_2\rangle$ in $\mathtt{PC}(r, w)$, we write $\langle u_1, t_1, v_1\rangle \lessdot \langle u_2, t_2, v_2\rangle$ if either $|u_1| < |u_2|$ or else $|u_1| = |u_2|$ and $t_1 \lessdot t_2$.

Theorem 1. $\mathtt{PC}(r, w)$ *is a finite and strict total order set.*

The finiteness ensures that $\mathtt{PC}(r, w)$ has the least (that is, the most prior) parse configuration, which we think of as representing the leftmost-longest matching. Notice that the least element may not exist for arbitrary (non-canonical) parse trees; e.g., for $\mathtt{1}*$ and the empty string, we would have an infinite decreasing sequence: $\mathtt{I}() \gtrdot \mathtt{I}(\mathtt{1}) \gtrdot \mathtt{I}(\mathtt{11}) \gtrdot \ldots$.

3 Enumerating Parse Trees via Position Automata

3.1 Correctly Nested Parenthesis Expressions

A *parenthesis expression* means a sequence consisting of parentheses, each of which is indexed by a position, or letters in Σ. A parenthesis expression α is called *correctly nested* if all parentheses in α are correctly nested. For each t in $\mathtt{CPT}(r)$, we assign, as its string representation, a correctly nested parenthesis expression $\Phi(t, p)$:

$$\Phi(1, p) = (_p\,)_p \qquad \Phi(\mathtt{R}(t), p) = (_p \Phi(t, p.2)\,)_p$$
$$\Phi(a, p) = (_p a\,)_p \quad (a \in \Sigma) \qquad \Phi(t_1 \cdot t_2, p) = (_p \Phi(t_1, p.1)\ \Phi(t_2, p.2)\,)_p$$
$$\Phi(\mathtt{L}(t), p) = (_p \Phi(t, p.1)\,)_p \qquad \Phi(\mathtt{I}(t_1, \ldots, t_n), p) = (_p \Phi(t_1, p.1) \ldots \Phi(t_n, p.1)\,)_p$$

For a parse configuration $\langle u, t, v\rangle$, we define $\Phi(\langle u, t, v\rangle) = u\ \Phi(t, \Lambda)\ v$. Note that the indexes in $\Phi(t, \Lambda)$ do not come from the parse tree t but from the AST of the underlying regular expression.

Φ is not injective in general; e.g., both $\Phi(\mathtt{I}(a), p)$ and $\Phi(\mathtt{L}(a), p)$ are $(_p (_{p.1} a)_{p.1})_p$. It is, however, injective if the domain is restricted to each $\mathtt{PC}(r, w)$. We refer to the image of $\mathtt{PC}(r, w)$ by Φ, i.e., $\{\Phi(c) \mid c \in \mathtt{PC}(r, w)\}$, as $\mathtt{PC}_\Phi(r, w)$.

Any expression in $\mathsf{PC}_\Phi(r)$ has a bounded nesting depth, which does not exceed the height of (the AST of) r indeed. This means that $\mathsf{PC}_\Phi(r)$ is a regular language, thereby, recognizable (or equivalently, enumerable) by an automaton. We will construct such automata in the next section.

Let α be an arbitrary parenthesis expression that is not necessarily in the range of the function Φ. Parenthesis expressions are sometimes "packetized" with regarding the occurrences of letters as separators. For any parenthesis expression α, which is always written in the form $\beta_0 a_1 \beta_1 a_2 \dots \beta_{n-1} a_n \beta_n$ where $a_1 \dots a_n$ are letters and $\beta_0 \dots \beta_n$ possibly empty sequences of parentheses, we call β_i $(0 \le i \le n)$ the i-th *frame* of α. Let $\alpha_0, \dots, \alpha_n$ and $\beta_0, \dots, \beta_n$ be the sequences of frames of α and β respectively. If α_k and β_k make the first distinction (that is, k is the greatest index such that $\alpha_i = \beta_i$ for any $0 \le i < k$), then the index k is called the *fork* of α and β.

3.2 Position NFAs with Augmented Transitions

A *position automaton with augmented transitions* (PAT, in short) is a 6-tuple which consists of (1) a finite set Σ of letters, (2) a finite set Q of *states*, (3) a finite set T of *tags*, (4) a subset Δ of $Q \times \Sigma \times Q \times T^*$, (5) an initial state $\mathfrak{q}_\wedge$ in Q and (6) a final state $\mathfrak{q}_\$$ in Q. For $a \in \Sigma$, Q_a denotes $\{q \in Q \mid \langle p, a, q, \tau\rangle \text{ for some } p \text{ and } \tau\}$. We call an element of Δ a *transition*.

For an input string $a_0 \dots a_{n-1}$, a sequence $q_0 \tau_0 \dots \tau_{n-1} q_n$ where $q_0 = \mathfrak{q}_\wedge$ is called a *path* if for any $0 \le i < n$ we have $\langle q_i, a_i, q_{i+1}, \tau_i\rangle \in \Delta$; $q_i \tau_i q_{i+1}$ $(0 \le i < n)$ is called the i-th step of the path. For a PAT $\langle \Sigma, Q, T, \Delta, \mathfrak{q}_\wedge, \mathfrak{q}_\$\rangle$ and an input string w, a *configuration* is a triple $\langle u, p, \alpha\rangle$ where u is a suffix of w, p a state in Q, and α a sequence of tags. For a transition $\delta = \langle p_1, a, p_2, \tau\rangle$ in Δ, we write $\langle u_1, p_1, \alpha_1\rangle \vdash_\delta \langle u_2, p_2, \alpha_2\rangle$ if $u_1 = a u_2$ and $\alpha_2 = \alpha_1 \tau a$. The *initial* configuration is $\langle w, \mathfrak{q}_\wedge, \varepsilon\rangle$ (where ε denotes the empty sequence), while a *finial* configuration is of the form $\langle \varepsilon, \mathfrak{q}_\$, \alpha\rangle$ for some α. We say a PAT M *accepts* an input string w, *yielding* a sequence α if there exists a sequence of configurations: $\langle w, \mathfrak{q}_\wedge, \varepsilon\rangle \vdash \cdots \vdash \langle \varepsilon, \mathfrak{q}_\$, \alpha\rangle$.

It might help to see a PAT as a sequential transducer. However, a PAT emits parenthesis expressions only for the sake of deciding priority of paths. Hence, a PAT has more similarity with Lurikari's *NFA with tagged transitions* (TNFA) [9] although the formulations are rather different; in a PAT, a transition is augmented with a sequence of tags, while TNFA allows at most one tag for each transition. Another difference is that a PAT is ε-transition free; indeed, a PAT is based on a position NFA, while a TNFA primary assumes a Thompson NFA as its basis, so that it has ε-transitions.

Let r be a regular expression, and p a position of (the AST of) r. The PAT $M(r, p)$ for r with respect to p is recursively constructed according to the structure of r as follows:

1. $M(1, p)$ is $\langle \Sigma, \{\mathfrak{q}_\wedge, \mathfrak{q}_\$\}, \{(_p,)_p\}, \{\langle \mathfrak{q}_\wedge, a, \mathfrak{q}_\$, (_p)_p\rangle \mid a \in \Sigma\}, \mathfrak{q}_\wedge, \mathfrak{q}_\$\rangle$.
2. For $a \in \Sigma$, $M(a, p)$ is $\langle \Sigma, \{\mathfrak{q}_\wedge, p, \mathfrak{q}_\$\}, \{(_p,)_p\}, \Delta, \mathfrak{q}_\wedge, \mathfrak{q}_\$\rangle$ where Δ consists of the transitions $\langle \mathfrak{q}_\wedge, a, p, (_p\rangle$ and $\langle p, b, \mathfrak{q}_\$,)_p\rangle$ for any $b \in \Sigma$.
3. If $M(r_i, p.i)$ is $\langle \Sigma, Q_i, T_i, \Delta_i, \mathfrak{q}_\wedge, \mathfrak{q}_\$\rangle$ for $i = 1, 2$, then $M(r_1 + r_2, p)$ is

$$\langle \Sigma, Q_1 \cup Q_2, T_1 \cup T_2 \cup \{(_p,)_p\}, [\Delta_1 \cup \Delta_2]_p, \mathfrak{q}_\wedge, \mathfrak{q}_\$\rangle.$$

4. If $M(r_i, p.i)$ is $\langle \Sigma, Q_i, T_i, \Delta_i, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$ for $i = 1, 2$, then $M(r_1 \cdot r_2, p)$ is

$$\langle \Sigma, Q_1 \cup Q_2, T_1 \cup T_2 \cup \{(_p,)_p\}, [\Delta_1 \cdot \Delta_2]_p, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle.$$

5. If $M(r, p.1)$ is $\langle \Sigma, Q, T, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$, then $M(r*, p)$ is $\langle \Sigma, Q, T \cup \{(_p,)_p\}, [\Delta*]_p, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$.

Finally, for $M(r, \Lambda) = \langle \Sigma, Q, T, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$, we define the PAT $M(r)$ for a regular expression r as $\langle \Sigma, Q, T, \Delta \cup \Delta_0, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$ where Δ_0 consists of the transitions $\langle \mathtt{q}_\wedge, a, \mathtt{q}_\wedge, \varepsilon \rangle$ for any $a \in \Sigma$ and $\langle \mathtt{q}_\$, a, \mathtt{q}_\$, \varepsilon \rangle$ for any $a \in \Sigma$.

In the above construction, $[\Delta]_p$, $\Delta_1 \cdot \Delta_2$ and $\Delta*$ are respectively given as follows:

$$\begin{aligned}
[\Delta]_p &= \{\langle q_1, a, q_2, \tau \rangle \in \Delta \mid q_1 \neq \mathtt{q}_\wedge, q_2 \neq \mathtt{q}_\$\} \cup \{\langle \mathtt{q}_\wedge, a, \mathtt{q}_\$, (_p\tau)_p \rangle \mid \langle \mathtt{q}_\wedge, a, \mathtt{q}_\$, \tau \rangle \in \Delta\} \\
&\quad \cup \{\langle \mathtt{q}_\wedge, a, q, \ (_p\tau \rangle \mid \langle \mathtt{q}_\wedge, a, q, \tau \rangle \in \Delta, q \neq \mathtt{q}_\$\} \\
&\quad \cup \{\langle q, a, \mathtt{q}_\$, \ \tau)_p \rangle \mid \langle q, a, \mathtt{q}_\$, \tau \rangle \in \Delta, q \neq \mathtt{q}_\wedge\} \\
\Delta_1 \cdot \Delta_2 &= \{\langle q_1, a, q_2, \tau \rangle \in \Delta_1 \mid q_2 \neq \mathtt{q}_\$\} \cup \{\langle q_1, a, q_2, \tau \rangle \in \Delta_2 \mid q_1 \neq \mathtt{q}_\wedge\} \\
&\quad \cup \{\langle q_1, a, q_2, \tau_1\tau_2 \rangle \mid \langle q_1, _, \mathtt{q}_\$, \tau_1 \rangle \in \Delta_1, \langle \mathtt{q}_\wedge, a, q_2, \tau_2 \rangle \in \Delta_2\} \\
\Delta* &= \Delta \cup \{\langle \mathtt{q}_\wedge, a, \mathtt{q}_\$, \varepsilon \rangle \mid a \in \Sigma\} \\
&\quad \cup \{\langle q_1, a, q_2, \tau_1\tau_2 \rangle \mid \langle q_1, _, \mathtt{q}_\$, \tau_1 \rangle \in \Delta, \langle \mathtt{q}_\wedge, a, q_2, \tau_2 \rangle \in \Delta, q_1 \neq \mathtt{q}_\wedge, q_2 \neq \mathtt{q}_\$\}
\end{aligned}$$

Fig. 1 shows the PAT $M(\texttt{(ab+a*)*})$ obtained by our construction, where the symbol $\square$ stands for arbitrary letters in Σ; that is, a transition with $\square$ actually represents several transitions obtained by replacing $\square$ with a letter in Σ.

Notice that the PATs constructed above requires a *look-ahead* symbol. We assume that Σ includes an extra symbol \$ for indicating the "end of string," and that any string given to a PAT $M(r)$ only has a trailing \$.

Let $\mathtt{PE}(M, w) = \{\alpha \mid \ M \text{ accepts } w\$ \text{ yielding } \alpha\$ \}$. The following theorem states that a PAT is capable of exactly enumerating any, and only canonical parse configurations:

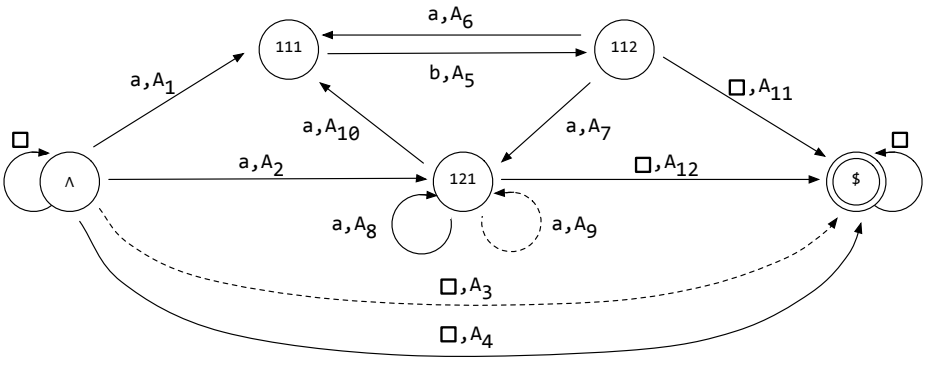
Theorem 2. $\mathtt{PC}_\Phi(r, w) = \mathtt{PE}(M(r), w)$.

4 Developing a Matching Algorithm

4.1 Basic Idea for Choosing the Most Prior Path

As mentioned before, our matching algorithm basically emulates the subset construction on the fly. The only but crucial difference is that we need to choose, in each step, the most prior one when paths converge on the same state. To spell out how to do this, consider *imaginary* stacks, one for each path. Along each path, opening parentheses are pushed on the stack in order of occurrence, and are eventually removed when the corresponding closing parentheses are encountered.

Consider two paths and the first step at which they are branching. At that moment, the stacks for these paths store exactly the same content, which, since opening parentheses occur in order of priority, corresponds to the closing parentheses that contribute to decide which path is prior to the other. To distinguish this content, we prepare a *bottom* pointer for each stack, which initially designates the top of the content (equivalently, the



A_1: (∧,(1,(11,(111
A_2: (∧,(1,(12,(121
A_3: (∧,)∧
A_4: (∧,(1,(12,)12,)1,)∧
A_5:)111,(112
A_6:)112,)11,)1,(1,(11,(111
A_7:)112,)11,)1,(1,(12,(121
A_8:)121,(121
A_9:)121,)12,)1,(1,(12,(121
A_{10}:)121,)12,)1,(1,(11,(111
A_{11}:)112,)11,)1,)∧
A_{12}:)121,)12,)1,)∧

Fig. 1. The position automata with augmented transitions generated from the regular expression (a·b+a*)*

bottom of the forthcoming parentheses) and decreases when the corresponding closing parentheses are found. Comparing the bottom pointers at each step allows us to know exactly when the corresponding closing parentheses appear.

Actually, the particular content of each stack does not matter since we already know that parentheses are correctly nested; what's really important is the minimum record of each bottom pointer that have ever achieved within the steps from the beginning to the current step. Moreover, since the index of a parenthesis indicates the value of the stack pointer when it is pushed, we no longer need the stack pointers.

This consideration allows us to develop a rather simple way of comparing the priority of paths only based on operations of the bottom pointers, without concerning each of particular parentheses at runtime. This idea is formalized in Section 4.2 below.

4.2 Formalization

First, we define the *height* of an opening parenthesis $(_p$ as $|p| + 1$ while the height of a closing parenthesis $)_p$ as p. Intuitively, the height of a parenthesis is the value of the (imaginary) stack pointer we discussed above.

For any sequences α and β of parentheses, $\alpha \sqcap \beta$ denotes the longest common prefix of α and β. For any sequence α of parentheses and a prefix α' of α, $\alpha \backslash \alpha'$ denotes the remaining sequence obtained by removing α' from α. In case $\alpha \backslash (\alpha \sqcap \beta)$ is non-empty, we denote the first element of $\alpha \backslash (\alpha \sqcap \beta)$ as α / β.

Algorithm 1. Match a string w against a regular expression r

```
1:  K := {q_∧}
2:  P[q][t] := −1 for any state q and tag t in the set T_cap of tags for captured subexpressions
3:  for all n := 0, ..., |w| do
4:      Read the next letter a in w$ (left to right)
5:      trans := effective_transitions(a)      // Choose transitions available for this step
6:      proceed_one_step(n, trans)             // Update D, P, K and B accordingly
7:      if K contains the final state q_$ then
8:          From K drop all q such that P[q][(_∧] > P[q_$][(_∧] or P[q][(_∧] = −1
9:          break if K contains only q_$
10:     end if
11: end for
12: if K contains the final state q_$ then
13:     Report SUCCESS and output P[q_$][t] for each t ∈ T_cap
14: else
15:     Report FAILURE
16: end if
```

Algorithm 2. `effective_transitions`(a)

```
1:  trans := ∅
2:  for all state q ∈ Q_a such that tag(p, q) ≠ ⊥ for some p ∈ K do
3:      K' := K\{p}
4:      for all state p' ∈ K' such that tag(p', q) ≠ ⊥ do
5:          ⟨ρ, ρ'⟩ := B[p][p']
6:          ρ := min{ρ, minsp(tag(p, q))};  ρ' := min{ρ', minsp(tag(p', q))};
7:          p := p' if ρ < ρ' or ρ = ρ' and D[p'][p] = 1
8:      end for
9:      Add ⟨p, q, tag(p, q)⟩ to trans
10: end for
11: return trans
```

Let α and β be parenthesis expressions whose frames are $\alpha_0, \ldots, \alpha_n$ and $\beta_0, \ldots, \beta_n$ respectively. Suppose that k is the fork (i.e., the index of the first different frames) of α and β. We define the *trace* of α with respect to β, written $\mathtt{tr}_\beta(\alpha)$, as a sequence $\rho_0, \ldots, \rho_n$ of integers as follows:

$$\rho_i = \begin{cases} -1 & (i < k) \\ \min\{\mathtt{lastsp}(\alpha_k \sqcap \beta_k), \mathtt{minsp}(\alpha_k \backslash (\alpha_k \sqcap \beta_k))\} & \\ \min\{\rho_{i-1}, \mathtt{minsp}(\alpha_i)\} & (i > k) \end{cases}$$

where $\mathtt{lastsp}(\gamma)$ denotes the height of the last (the rightmost) parenthesis in γ or else 0 if γ is empty, while $\mathtt{minsp}(\gamma)$ is the minimal height of the parentheses in γ or else 0 if γ is empty. Intuitively, The i-th value ρ_i of $\mathtt{tr}_\beta(\alpha)$ ($i > k$ where k is the fork) means the minimal record of the bottom pointer for α within the steps from the fork to the i-th step. The negative number -1 just means that the bottom pointer is not yet set. We denote the initial value $\min\{\mathtt{lastsp}(\alpha \sqcap \beta), \mathtt{minsp}(\alpha \backslash (\alpha \sqcap \beta))\}$ as $\mathtt{bp0}_\beta(\alpha)$.

Algorithm 3. `proceed_one_step`$(n, trans)$

$K := \{q \mid \langle \_, q, \_ \rangle \in trans\}$
$D' := D$; $P' := P$; $B' := B$
for all $\langle p, q, \alpha \rangle, \langle p', q', \alpha' \rangle \in trans$ such that $q \lessdot q'$ **do**
  if $p = p'$ **then**
    $D[q][q'] := 1$ **if** $\alpha \sqsubset \alpha'$; $D[q][q'] := -1$ **if** $\alpha' \sqsubset \alpha$
    $\rho := \mathtt{bp0}_{\alpha'}(\alpha)$; $\rho' := \mathtt{bp0}_{\alpha}(\alpha')$
  else
    $D[q][q'] := D'[p][p']$
    $\langle \rho, \rho' \rangle := B'[p][p']$
    $\rho := \min\{\rho, \mathtt{minsp}(\alpha)\}$; $\rho' := \min\{\rho', \mathtt{minsp}(\alpha')\}$;
  end if
  $D[q][q'] := 1$ **if** $\rho > \rho'$; $D[q][q'] := -1$ **if** $\rho < \rho'$; $D[q'][q] := -D[q][q']$
  $B[q][q'] := \langle \rho, \rho' \rangle$; $B[q'][q] := \langle \rho', \rho \rangle$
end for
for all $\langle p, q, \alpha \rangle \in trans$ **do**
  $P[q] := P'[p]$
  $P[q][t] := n$ **for all** $t \in T_{\mathrm{cap}} \cap \alpha$
end for

For $\mathtt{tr}_{\alpha'}(\alpha) = \rho_0, \ldots, \rho_n$ and $\mathtt{tr}_{\alpha}(\alpha') = \rho'_0, \ldots, \rho'_n$, we write $\mathtt{tr}_{\alpha'}(\alpha) \lessdot \mathtt{tr}_{\alpha}(\alpha')$ if $\rho_i > \rho'_i$ for the least i such that $\rho_j = \rho'_j$ for any $j > i$. For frames α and α', we write $\alpha \sqsubset \alpha'$, if the following conditions, (1) and (2), hold for some position p; (1) there exists $\alpha/\alpha' = (_p$; (2) if there exists $\alpha'/\alpha = (_q$ for some q then $p \lessdot q$.

Definition 3. *Let α and β be parenthesis expressions. We say α is* prior to *β, written $\alpha \lessdot \beta$, if either $\mathtt{tr}_{\beta}(\alpha) \lessdot \mathtt{tr}_{\alpha}(\beta)$ or else $\mathtt{tr}_{\beta}(\alpha) = \mathtt{tr}_{\alpha}(\beta)$ and $\alpha' \sqsubset \beta'$ where α' and β' are the k-th frames of α and β respectively and k is the fork of α and β.*

The order we have just defined above is essentially the same as the priority order on parse configurations:

Theorem 3. *$\langle \mathsf{PC}_{\Phi}(r, w), \lessdot \rangle$ is an order set isomorphic to $\langle \mathsf{PC}(r, w), \lessdot \rangle$.*

4.3 Algorithm

Based on the above discussion, we provide a regular expression matching algorithm. Algorithm 1 is the main routine of the algorithm, which takes, apart from a PAT $M(r)$ built from a regular expression r, an input string $w\$$ then answers whether w matches r or not. If the matching succeeds, the algorithm tells us the positions of substrings captured by subexpressions.

Throughout the entire algorithm, a couple of global variables are maintained: K, B, D and P. K stores the set of current states. Let α and β be the paths getting to states p and q respectively. Then, $B[p][q]$ stores the minimal record of the bottom pointer for α and β. $D[p][q]$ remembers which path is currently prior to the other; this decision might be overridden later. $P[p][t]$, where t is a tag (i.e., a parenthesis), is the last step number in α at which t occurs.

The main routine invokes two other subroutines; Algorithm 2 takes a letter a of the input string then returns a set of transitions actually used for that step, pruning less prior, thus ineffective, transitions, while Algorithm 3 updates the values of the global variables for the next step of computation.

Algorithm 2 assumes that a given automaton has been preprocessed so that we have at most one transition $\langle p, q, a, \tau \rangle$ for any triple $\langle p, q, a \rangle$ such that $p, q \in Q$ and $a \in \Sigma$ by removing some (less prior) transitions. The following proposition, which immediately follows by Def. 3, justifies this:

Proposition 1. *Let α and β be parenthesis expressions whose frames are the same except for the k-th frames, say α' and β', for some k. We have $\alpha \lessdot \beta$ if either $\mathtt{bp0}_{\beta'}(\alpha') > \mathtt{bp0}_{\alpha'}(\beta')$ or else $\mathtt{bp0}_{\beta'}(\alpha') = \mathtt{bp0}_{\alpha'}(\beta')$ and $\alpha' \sqsubset \beta'$.*

For the automaton in Fig. 1, the prepossessing removes the transitions with A_3 and A_9 drawn in dashed lines. In Alg. 2, $\mathtt{tag}(p, q)$ denotes the sequence α of tags such that $\langle p, a, q, \alpha \rangle \in \Delta$ for some $a \in \Sigma$, or $\bot$ if such a transition does not exist (such α is determined uniquely if it exists).

Our algorithm answers the latest positions of captured substrings[1] consistent with those given by the most prior parenthesis expression:

Theorem 4. *Let α be the least parenthesis expression in $\mathtt{PC}_\Phi(r, w)$ and $\alpha_0, \ldots, \alpha_n$ its frames. When the algorithm terminates, $P[\mathtt{q}_\$][t]$ gives the greatest i ($\leq n$) such that α_i contains t for any $t \in T_{\mathrm{cap}}$ that occur in α; $P[\mathtt{q}_\$][t] = -1$ for the other $t \in T_{\mathrm{cap}}$.*

We now consider the runtime cost of our algorithm. We need not count the computational cost for $\mathtt{bp}(\alpha)$, $\mathtt{bp0}_\beta(\alpha)$ and $\alpha \sqsubset \beta$ because they can computed, in advance, at compile time (before a particular input string is given). The number of transitions to be examined in $\mathtt{effective\_transitions}(a)$ does not exceed $|K| \cdot |Q_a|$ and we have $|K| \leq |Q_{a'}|$ for the letter a' processed in the previous iteration in Alg. 1. Since $|Q_a|$ (resp. $|Q_{a'}|$) is the number of occurrences of the letter a (resp. a') in the given regular expression plus 2, the time complexity of Alg. 2 is $O(n^2)$, where n is the number of occurrences of the most frequently used letter in the given regular expression and, likewise, we obtain $O(n(n + c))$ for Alg. 3, where c is the number of subexpressions with which we want to capture substrings. Therefore, for the length m of a input string, the time complexity of the whole algorithm is given as follows:

Theorem 5. *The time complexity of the algorithm at runtime is $O(mn(n + c))$.*

5 Related Work

The idea of realizing regular expression matching by emulating subset construction at runtime goes back to early days; see [2] for the history. For the idea of using tags in automata we are indebted to the study [9], in which Laurikari have introduced NFA with

[1] In [5], capturing within a previous iteration should be reset once; e.g, the matching *aba* to $(a(b)*)*$ should result in $P[\mathtt{q}_\$][(_{121}] = P[\mathtt{q}_\$][)_{121}] = -1$, while our algorithm currently reports $P[\mathtt{q}_\$][(_{121}] = 1$, $P[\mathtt{q}_\$][)_{121}] = 2$. This subtle gap could be resolved without affecting the order of the computational cost by replacing P with more appropriate data structures: e.g., trees.

tagged transitions (TNFA), the basis of TRE library, in order to formulate the priority of paths. Unfortunately, it is somewhat incompatible with today's interpretation [5] as for the treatment of repetition.

Dubé and Feeley have given a way of generating the entire set of parse trees from regular expressions [4] by using a grammatical representation. Frisch and Cardelli have also considered an ordering on parse trees [6], which is completely different from ours since they focus on the greedy, or first match, semantics. Their notion of "non-problematic value" is similar to, but stronger than, our canonical parse trees. They also focus on a problem of ε-transition loop, which does not occur in our case since we are based on position automata. Vansummeren [11] have also given a stable theoretical framework for the leftmost-longest matching, although capturing inside repetitions is not considered.

An implementation taking a similar approach to ours is Kuklewicz's Haskell TDFA library [8]. Although it is also based on position automata, the idea of using *orbit tags* for the comparison of paths is completely different from our approach. Another similar one is Google's new regular expression library called RE2 [3] which has come out just before we finish the preparation of this paper. Unlike ours, RE2 follows the greedy semantics rather than the leftmost-longest semantics. TRE, TDFA, RE2 and our algorithm are all based on automata, so that, while their scope is limited to regular expressions without *back references*, they all enable of avoiding the computational explosion.

Acknowledgments. The authors are grateful to anonymous reviewers for valuable comments.

References

1. The Open Group Base Specification Issue 6 IEEE Std 1003.1 2004 Edition (2004), `http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html`
2. Cox, R.: Regular Expression Matching Can Be Simple and Fast (2007), `http://swtch.com/~rsc/regexp/regexp1.html`
3. Cox, R.: Regular Expression Matching in the Wild (2010), `http://swtch.com/~rsc/regexp/regexp3.html`
4. Dubé, D., Feeley, M.: Efficiently Building a Parse Tree from a Regular Expression. Acta Infomatica 37(2), 121–144 (2000)
5. Fowler, G.: An Iterpretation of the POSIX Regex Standard (2003), `http://www2.research.att.com/~gsf/testregex/re-interpretation.html`
6. Frisch, A., Cardelli, L.: Greedy Regular Expression Matching. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 618–629. Springer, Heidelberg (2004)
7. Glushkov, V.M.: The Abstract Theory of Automata. Russian Mathematical Surveys 16(5), 1–53 (1961)
8. Kuklewicz, C.: Regular Expressions: Bounded Space Proposal (2007), `http://www.haskell.org/haskellwiki/Regular_expressions/Bounded_space_proposal`
9. Laurikari, V.: Efficient Submatch Addressing for Regular Expressions. Master's thesis, Helsinki University of Technology (2001)
10. McNaughton, R., Yamada, H.: Regular Expressions and State Graphs for Automata. IEEE Transactions on Electronic Computers 9, 39–47 (1960)
11. Vansummeren, S.: Type Inference for Unique Pattern Matching. ACM Transactions on Programming Languages and Systems 28(3), 389–428 (2006)

A Polynomial Time Match Test for Large Classes of Extended Regular Expressions

Daniel Reidenbach and Markus L. Schmid*

Department of Computer Science, Loughborough University,
Loughborough, Leicestershire, LE11 3TU, United Kingdom
{D.Reidenbach,M.Schmid}@lboro.ac.uk

Abstract. In the present paper, we study the match test for extended regular expressions. We approach this NP-complete problem by introducing a novel variant of two-way multihead automata, which reveals that the complexity of the match test is determined by a hidden combinatorial property of extended regular expressions, and it shows that a restriction of the corresponding parameter leads to rich classes with a polynomial time match test. For presentational reasons, we use the concept of pattern languages in order to specify extended regular expressions. While this decision, formally, slightly narrows the scope of our results, an extension of our concepts and results to more general notions of extended regular expressions is straightforward.

1 Introduction

Regular expressions are compact and convenient devices that are widely used to specify regular languages, e. g., when searching for a pattern in a string. In order to overcome their limited expressive power while, at the same time, preserving their desirable compactness, their definition has undergone various modifications and extensions in the past decades. These amendments have led to several competing definitions, which are collectively referred to as *extended regular expressions* (or: *REGEX* for short). Hence, today's text editors and programming languages (such as Java and Perl) use individual notions of (extended) regular expressions, and they all provide so-called *REGEX engines* to conduct a *match test*, i. e., to compute the solution to the membership problem for any language given by a REGEX and an arbitrary string. While the introduction of new features of extended regular expressions have frequently not been guided by theoretically sound analyses, recent studies have led to a deeper understanding of their properties (see, e. g., Câmpeanu et al. [3]).

A common feature of extended regular expressions not to be found in the original definition is the option to postulate that each word covered by a specific REGEX must contain a variable substring at several recurrent positions (so-called *backreferences*). Thus, they can be used to specify a variety of non-regular languages (such as the language of all words w that satisfy $w = xx$ for arbitrary

* Corresponding author.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 241–250, 2011.

words x), and this has severe consequences on the complexity of their basic decision problems. In particular, their vital membership problem (i. e., in other words, the match test) is NP-complete (see Aho [1]). REGEX engines commonly use more or less sophisticated *backtracking* algorithms over extensions of NFA in order to perform the match test (see Friedl [5]), often leading even for rather small inputs to a practically unbearable runtime. Therefore, it is a worthwhile task to investigate alternative approaches to this important problem and to establish large classes of extended regular expressions with a polynomial-time match test.

It is the purpose of this paper to propose and study such an alternative method. In order to keep the technical details reasonably concise we do not directly use a particular REGEX definition, but we consider a well-established type of formal languages that, firstly, is defined in a similar yet simpler manner, secondly, is a proper subclass of the languages generated by REGEX and, thirdly, shows the same properties with regard to the membership problem: the *pattern languages* as introduced by Angluin [2]; our results on pattern languages can then directly be transferred to the corresponding class of REGEX. In this context, a *pattern* α is a finite string that consists of *variables* and *terminal symbols* (taken from a fixed alphabet Σ), and its language is the set of all words that can be derived from α when substituting arbitrary words over Σ for the variables. For example, the language L generated by the pattern $\alpha := x_1\mathtt{a}x_2\mathtt{b}x_1$ (with variables x_1, x_2 and terminal symbols $\mathtt{a}, \mathtt{b}$) consists of all words with an arbitrary prefix u, followed by the letter $\mathtt{a}$, an arbitrary word v, the letter $\mathtt{b}$ and a suffix that equals u. Thus, $w_1 := \mathtt{aaabbaa}$ is contained in L, whereas $w_2 := \mathtt{baaba}$ is not.

In the definition of pattern languages, the option of using several occurrences of a variable exactly corresponds to the backreferences in extended regular expressions, and therefore the membership problem for pattern languages captures the essence of what is computationally complex in the match test for REGEX. Thus, it is not surprising that the membership problem for pattern languages is also known to be NP-complete (see Angluin [2] and Jiang et al. [10]). Furthermore, Ibarra et al. [9] point out that the membership problem for pattern languages is closely related to the solvability problem for certain Diophantine equations. More precisely, for any word w and for any pattern α with m terminal symbols and n different variables, w can only be contained in the language generated by α if there are numbers s_i (representing the lengths of the substitution words for the variables x_i) such that $|w| = m + \sum_{i=1}^{n} a_i s_i$ (where a_i is the number of occurrences of x_i in α and $|w|$ stands for the *length* of w). Thus, the membership test needs to implicitly solve this NP-complete problem, which is related to *Integer Linear Programming* problems (see the references in [9]) and the *Money-Changing Problem* (see Guy [6]). All these insights into the complexity of the membership problem do not depend on the question of whether the pattern contains any terminal symbols. Therefore, we can safely restrict our considerations to so-called *terminal-free* pattern languages (generated by patterns that consist of variables only); for this case, NP-completeness of the membership problem has indirectly been established by Ehrenfeucht and Rozenberg [4]. This

restriction again improves the accessibility of our technical concepts, without causing a loss of generality.

As stated above, these results on the complexity of the problem (and the fact that probabilistic solutions might often be deemed inappropriate for it) motivate the search for large subclasses with efficiently solvable membership problem and for suitable concepts realising the respective algorithms. Rather few such classes are known to date. They either restrict the number of *different* variables in the patterns to a fixed number k (see Angluin [2], Ibarra et al. [9]), which is an obvious option and leads to a time complexity of $\mathrm{O}(n^k)$, or they restrict the number of *occurrences* of each variable to 1 (see Shinohara [11]), which turns the resulting pattern languages into regular languages.

In the present paper, motivated by Shinohara's [12] *non-cross* pattern languages, we introduce major classes of pattern languages (and, hence, of extended regular expressions) with a polynomial-time membership problem that do not show any of the above limitations. Thus, the corresponding patterns can have any number of variables with any number of occurrences; instead, we consider a rather subtle parameter, namely the *distance* several occurrences of any variable x may have in a pattern (i. e., the maximum number of different variables separating any two consecutive occurrences of x). We call this parameter the *variable distance* vd of a pattern, and we demonstrate that, for the class of all patterns with $\mathrm{vd} \leq k$, the membership problem is solvable in time $\mathrm{O}(n^{k+4})$. Referring to the proximity between the subject of our paper and the solvability problem of the equation $|w| = m + \sum_{i=1}^{n} a_i s_i$ described above (which does not depend on the order of variables in the patterns, but merely on their numbers of occurrences), we consider this insight quite remarkable, and it is only possible since this solvability problem is *weakly* NP-complete (i. e. there exist pseudo-polynomial time algorithms). We also wish to point out that, in terms of our concept, Shinohara's non-cross patterns correspond to those patterns with $\mathrm{vd} = 0$.

We prove our main result by introducing the concept of a *Janus automaton*, which is a variant of a two-way two-head automaton (see Ibarra [7]), amended by the addition of a number of counters. Janus automata are algorithmic devices that are tailored to performing the match test for pattern languages, and we present a systematic way of constructing them. While an intuitive use of a Janus automaton assigns a distinct counter to each variable in the corresponding pattern α, we show that in our advanced construction the number of different counters can be limited by the variable distance of α. Since the number of counters is the main element determining the complexity of a Janus automaton, this yields our main result. An additional effect of the strictness of our approach is that we can easily discuss its quality in a formal manner, and we can show that, based on a natural assumption on how Janus automata operate, our method leads to an automaton with the smallest possible number of counters. Furthermore, it is straightforward to couple our Janus automata with ordinary finite automata in order to expand our results to more general classes of extended regular expressions, e. g., those containing terminal symbols or imposing regular restrictions to the sets of words variables can be substituted with.

In order to validate our claim that the variable distance is a crucial parameter contributing to the complexity of the match test, and to examine whether our work – besides its theoretical value – might have any practical relevance, some instructive tests have been performed.[1] They compare a very basic Java implementation of our Janus automata with the original REGEX engine included in Java. With regard to the former objective, the test results suggest that our novel notion of a variable distance is indeed a crucial (and, as briefly mentioned above, rather counter-intuitive) parameter affecting the complexity of the match test for both our Janus-based algorithm and the established backtracking method. Concerning the latter goal, we can observe that our non-optimised implementation, on average, considerably outperforms Java's REGEX engine. We therefore conclude that our approach might also be practically worthwhile.

2 Definitions

Let $\mathbb{N} := \{0, 1, 2, 3, \ldots\}$. For an arbitrary alphabet A, a *string* (*over* A) is a finite sequence of symbols from A, and ε stands for the *empty string*. The symbol A^+ denotes the set of all nonempty strings over A, and $A^* := A^+ \cup \{\varepsilon\}$. For the *concatenation* of two strings w_1, w_2 we write $w_1 \cdot w_2$ or simply $w_1 w_2$. We say that a string $v \in A^*$ is a *factor* of a string $w \in A^*$ if there are $u_1, u_2 \in A^*$ such that $w = u_1 \cdot v \cdot u_2$. The notation $|K|$ stands for the size of a set K or the length of a string K; the term $|w|_a$ refers to the number of occurrences of the symbol a in the string w.

For any alphabets A, B, a *morphism* is a function $h : A^* \to B^*$ that satisfies $h(vw) = h(v)h(w)$ for all $v, w \in A^*$. Let Σ be a (finite) alphabet of so-called *terminal symbols* and X an infinite set of *variables* with $\Sigma \cap X = \emptyset$. We normally assume $X := \{x_1, x_2, x_3, \ldots\}$. A *pattern* is a nonempty string over $\Sigma \cup X$, a *terminal-free pattern* is a nonempty string over X and a *word* is a string over Σ. For any pattern α, we refer to the set of variables in α as $\mathrm{var}(\alpha)$. We shall often consider a terminal-free pattern in its variable factorisation, i. e. $\alpha = y_1 \cdot y_2 \cdot \ldots \cdot y_n$ with $y_i \in \{x_1, x_2, \ldots, x_m\}$, $1 \leq i \leq n$ and $m = |\mathrm{var}(\alpha)|$. A morphism $\sigma : (\Sigma \cup X)^* \to \Sigma^*$ is called a *substitution* if $\sigma(a) = a$ for every $a \in \Sigma$.

We define the *pattern language* of a terminal-free pattern α by $L_\Sigma(\alpha) := \{\sigma(\alpha) \mid \sigma : X^* \to \Sigma^* \text{ is a substitution}\}$. Note, that these languages, technically, are terminal-free E-pattern languages (see Jiang et al. [10]). We ignore the case where a variable occurs just once, as then $L_\Sigma(\alpha) = \Sigma^*$.

The problem to decide for a given pattern α and a given word $w \in \Sigma^*$ whether $w \in L_\Sigma(\alpha)$ is called the *membership problem*.

3 Janus Automata

In the present section we introduce a novel type of automata, the so-called Janus automata, that are tailored to solving the membership problem for pattern languages. To this end, we combine elements of two-way multihead finite automata (see, e. g., Ibarra [7]) and counter machines (see, e. g., Ibarra [8]).

[1] Tests and source code are available at `http://www-staff.lboro.ac.uk/~coms10/`

A *Janus automaton* (or JFA(k) for short) is a two-way 2-head automaton with k restricted counters, $k \in \mathbb{N}$. More precisely, it is a tuple $M := (k, Q, \Sigma, \delta, q_0, F)$, where Σ is an *input alphabet*, δ is a *transition function*, Q is a set of *states*, $F \subseteq Q$ is a set of *final states*, and $q_0 \in Q$ is the *initial state*. In each step of the computation the automaton M provides a distinct *counter bound* for each counter. The *counter values* can only be incremented or left unchanged and they count strictly modulo their counter bound, i. e. once a counter value has reached its counter bound, a further incrementation forces the counter to start at counter value 1 again. Depending on the current state, the currently scanned input symbols and on whether the counters have reached their bounds, the transition function determines the next state, the input head movements and the counter instructions. In addition to the counter instructions of incrementing and leaving the counter unchanged it is also possible to reset a counter. In this case, the counter value is set to 0 and a new counter bound is nondeterministically guessed. Furthermore, we require the first input head to be always positioned to the left of the second input head, so there are a well-defined *left* and *right head*. Therefore, we call this automata model a "Janus" automaton. Any string $\&w\$$, where $w \in \Sigma^*$ and the symbols $\&, \$$ (referred to as *left* and *right endmarker*, respectively) are not in Σ, is an *input* to M. Initially, the *input tape* stores some input, M is in state q_0, all counter bounds and counter values are 0 and both input heads scan &. The word w is accepted by M if and only if it is possible for M to reach an accepting state by succesively applying the transition function. For any Janus automaton M let $L(M)$ denote the set of words accepted by M.

JFA(k) are nondeterministic automata, but their nondeterminism differs from that of common nondeterministic finite automata. The only nondeterministic step a Janus automaton is able to perform consists in guessing a new counter bound for some counter. Once a new counter bound is guessed, the previous one is lost. Apart from that, each transition, i. e. entering a new state, moving the input heads and giving instructions to the counters, is defined completely deterministically by δ.

The vital point of a JFA(k) computation is then, of course, that the automaton is only able to save exactly k (a constant number, not depending on the input word) different numbers at a time. For a JFA(k) M, the number k shall be the crucial number for the complexity of the *acceptance problem* (*for* M), i. e. to decide, for a given word w, whether $w \in L(M)$.

4 Janus Automata for Pattern Languages

In this chapter, we demonstrate how Janus automata can be used for recognising pattern languages. More precisely, for an arbitrary terminal-free pattern α, we construct a JFA(k) M satisfying $L(M) = L_\Sigma(\alpha)$. Before we move on to a formal analysis of this task, we discuss the problem of deciding whether $w \in L_\Sigma(\alpha)$ for given α and w, i. e. the membership problem, in an informal way.

Let $\alpha = y_1 \cdot y_2 \cdot \ldots \cdot y_n$ be a terminal-free pattern with $m := |\operatorname{var}(\alpha)|$, and let $w \in \Sigma^*$ be a word. The word w is an element of $L_\Sigma(\alpha)$ if and only if there exists

a factorisation $w = u_1 \cdot u_2 \cdot \ldots \cdot u_n$ such that $u_j = u_{j'}$ for all $j, j' \in \{1, 2, \ldots, |\alpha|\}$ with $y_j = y_{j'}$. Thus, a way to solve the membership problem is to initially guess m numbers $\{l_1, l_2, \ldots, l_m\}$, then, if possible, to factorise $w = u_1 \cdot \ldots \cdot u_n$ such that $|u_j| = l_i$ for all j with $y_j = x_i$ and, finally, to check whether $u_j = u_{j'}$ is satisfied for all $j, j' \in \{1, 2, \ldots, |\alpha|\}$ with $y_j = y_{j'}$. A JFA(m) can perform this task by initially guessing m counter bounds, which can be interpreted as the lengths of the factors. The two input heads can be used to check if this factorisation has the above described properties. However, the number of counters that are then required directly depends on the number of variables, and the question arises if this is always necessary. The next step is to formalise and generalise the way of constructing a JFA(k) for arbitrary pattern languages.

Definition 1. *Let $\alpha := y_1 \cdot y_2 \cdot \ldots \cdot y_n$ be a terminal-free pattern, and let $n_i := |\alpha|_{x_i}$ for each $x_i \in \mathrm{var}(\alpha)$. The set $\mathrm{varpos}_i(\alpha)$ is the set of all positions j satisfying $y_j = x_i$. Let furthermore $\Gamma_i := ((l_1, r_1), (l_2, r_2), \ldots, (l_{n_i-1}, r_{n_i-1}))$ with $(l_j, r_j) \in \mathrm{varpos}_i(\alpha)^2$ and $l_j < r_j$, $1 \leq j \leq n_i - 1$. The sequence Γ_i is a* matching order for x_i in α *if and only if the graph $(\mathrm{varpos}_i(\alpha), \Gamma_i')$ is connected, where $\Gamma_i' := \{(l_1, r_1), (l_2, r_2), \ldots, (l_{n_i-1}, r_{n_i-1})\}$. The elements $m_j \in \mathrm{varpos}_i(\alpha)^2$ of a matching order $(m_1, m_2, \ldots, m_k)$ are called* matching positions.

We illustrate Definition 1 by the example pattern $\beta := x_1 \cdot x_2 \cdot x_1 \cdot x_2 \cdot x_3 \cdot x_2 \cdot x_3$. Possible matching orders for x_1, x_2 and x_3 in β are given by $((1, 3))$, $((2, 4), (4, 6))$ and $((5, 7))$, respectively. To obtain a matching order for a pattern α we simply combine matching orders for all $x \in \mathrm{var}(\alpha)$:

Definition 2. *Let α be a terminal-free pattern with $m := |\mathrm{var}(\alpha)|$ and, for all i with $1 \leq i \leq m$, $n_i := |\alpha|_{x_i}$ and let $(m_{i,1}, m_{i,2}, \ldots, m_{i,n_i-1})$ be a matching order for x_i in α. The tuple $(m_1, m_2, \ldots, m_k)$ is a* complete matching order for α *if and only if $k = \sum_{i=1}^{m} n_i - 1$ and, for all i, j_i, $1 \leq i \leq m$, $1 \leq j_i \leq n_i - 1$, there is a j', $1 \leq j' \leq k$, with $m_{j'} = m_{i,j_i}$.*

With respect to our example pattern β this means that any sequence of the matching positions in $\{(1, 3), (2, 4), (4, 6), (5, 7)\}$ is a complete matching order for β. As pointed out by the following lemma, the concept of a complete matching order can be used to solve the membership problem.

Lemma 1. *Let $\alpha = y_1 \cdot y_2 \cdot \ldots \cdot y_n$ be a terminal-free pattern and $((l_1, r_1), (l_2, r_2), \ldots, (l_k, r_k))$ a complete matching order for α. Let w be an arbitrary word in some factorisation $w = u_1 \cdot u_2 \cdot \ldots \cdot u_n$. If $u_{l_j} = u_{r_j}$ for each j with $1 \leq j \leq k$, then $u_j = u_{j'}$ for all $j, j' \in \{1, 2, \ldots, |\alpha|\}$ with $y_j = y_{j'}$.*

Let $\alpha = y_1 \cdot y_2 \cdot \ldots \cdot y_n$ be a terminal-free pattern and let w be an arbitrary word in some factorisation $w = u_1 \cdot u_2 \cdot \ldots \cdot u_n$. According to the previous lemma, we may interpret a complete matching order as a list of instructions specifying how the factors u_i, $1 \leq i \leq n$, can be compared in order to check if $u_j = u_{j'}$ for all $j, j' \in \{1, 2, \ldots, |\alpha|\}$ with $y_j = y_{j'}$, which is of course characteristic for $w \in L_\Sigma(\alpha)$. With respect to the complete matching order $((4, 6), (1, 3), (2, 4), (5, 7))$ for the example pattern β, we apply Lemma 1 in the following way. If a word

$w \in \Sigma^*$ can be factorised into $w = u_1 \cdot u_2 \cdot \ldots \cdot u_7$ such that $u_4 = u_6$, $u_1 = u_3$, $u_2 = u_4$ and $u_5 = u_7$ then $w \in L_\Sigma(\beta)$. These matching instructions given by a complete matching order can be carried out by using two pointers, or input heads, moving over the word w.

Let (l', r') and (l, r) be two consecutive matching positions. It is possible to perform the comparison of factors $u_{l'}$ and $u_{r'}$ by positioning the left head on the first symbol of $u_{l'}$, the right head on the first symbol of $u_{r'}$ and then moving them simultaneously over these factors from left to right, checking symbol by symbol if these factors are identical. Now the left head, located at the first symbol of factor $u_{l'+1}$, has to be moved to the first symbol of factor u_l. If $l' < l$, then it is sufficient to move it over all the factors $u_{l'+1}, u_{l'+2}, \ldots, u_{l-1}$. If, on the other hand, $l < l'$, then the left head has to be moved to the left, thus over the factors $u_{l'}$ and u_l as well. Furthermore, as we want to apply these ideas to Janus automata, the heads must be moved in a way that the left head is always located to the left of the right head. The following definition shall formalise these ideas.

Definition 3. *Let $((l_1, r_1), (l_2, r_2), \ldots, (l_k, r_k))$ be a complete matching order for a terminal-free pattern α and let $l_0 := r_0 := 0$. For all j, j', $1 \leq j < j' \leq |\alpha|$ we define $g(j, j') := (j+1, j+2, \ldots, j'-1)$ and $g(j', j) := (j', j'-1, \ldots, j)$. For each i with $1 \leq i \leq k$ we define $D_i^\lambda := ((p_1, \lambda), (p_2, \lambda), \ldots, (p_{k_1}, \lambda))$ and $D_i^\rho := ((p'_1, \rho), (p'_2, \rho), \ldots, (p'_{k_2}, \rho))$, where $(p_1, p_2, \ldots, p_{k_1}) := g(l_{i-1}, l_i)$, $(p'_1, p'_2, \ldots, p'_{k_2}) := g(r_{i-1}, r_i)$ and λ, ρ are constant markers. Now let $D'_i := ((s_1, \mu_1), (s_2, \mu_2), \ldots, (s_{k_1+k_2}, \mu_{k_1+k_2}))$, with $s_j \in \{p_1, \ldots, p_{k_1}, p'_1, \ldots, p'_{k_2}\}$, $\mu_j \in \{\lambda, \rho\}$, $1 \leq j \leq k_1 + k_2$, be a tuple containing exactly the elements of D_i^λ and D_i^ρ such that the relative orders of the elements in D_i^λ and D_i^ρ are preserved. Furthermore, for each j, $1 \leq j \leq k_1 + k_2$, $q_j \leq q'_j$ needs to be satisfied, where $q_j := l_{i-1}$ if $\mu_{j'} = \rho$, $1 \leq j' \leq j$, and $q_j := \max\{j' \mid 1 \leq j' \leq j, \mu_{j'} = \lambda\}$ else, analogously, $q'_j := r_{i-1}$ if $\mu_{j'} = \lambda$, $1 \leq j' \leq j$, and $q'_j := \max\{j' \mid 1 \leq j' \leq j, \mu_{j'} = \rho\}$ else. Now we append the two elements (r_i, ρ), (l_i, λ) in exactly this order to the end of D'_i and obtain D_i. Finally, the tuple $(D_1, D_2, \ldots, D_k)$ is called a* Janus operating mode for α (derived from the complete matching order $((l_1, r_1), (l_2, r_2), \ldots, (l_k, r_k))$). *By $\overline{D_i}$, we denote the tuple D_i without the markers, i. e., if $D_i = ((p_1, \mu_1), \ldots, (p_n, \mu_n))$ with $\mu_j \in \{\lambda, \rho\}$, $1 \leq j \leq n$, then $\overline{D_i} := (p_1, p_2, \ldots, p_n)$.*

We recall once again the example $\beta := x_1 \cdot x_2 \cdot x_1 \cdot x_2 \cdot x_3 \cdot x_2 \cdot x_3$. According to Definition 3 we consider the tuples D_i^λ and D_i^ρ with respect to the complete matching order $((4,6),(1,3),(2,4),(5,7))$ for β. We omit the markers λ and ρ for a better presentation. The tuples D_i^λ are given by $D_1^\lambda = (1,2,3)$, $D_2^\lambda = (4,3,2,1)$, $D_3^\lambda = ()$ and $D_4^\lambda = (3,4)$. The tuples D_i^ρ are given by $D_1^\rho = (1,2,\ldots,5)$, $D_2^\rho = (6,5,4,3)$, $D_3^\rho = ()$ and $D_4^\rho = (5,6)$. Therefore, $\Delta_\beta := (D_1, D_2, D_3, D_4)$ is a possible Janus operating mode for β derived from $((4,6),(1,3),(2,4),(5,7))$, where $D_1 = ((1,\rho), (1,\lambda), (2,\rho), (2,\lambda), (3,\rho), (3,\lambda), (4,\rho), (5,\rho), (6,\rho), (4,\lambda))$, $D_2 = ((4,\lambda), (3,\lambda), \ldots, (1,\lambda), (6,\rho), (5,\rho), \ldots, (3,\rho), (3,\rho), (1,\lambda))$, $D_3 = ((4,\rho), (2,\lambda))$, $D_4 = ((3,\lambda), (4,\lambda), (5,\rho), (6,\rho), (7,\rho), (5,\lambda))$.

We shall see that it is possible to transform a Janus operating mode for any pattern directly into a Janus automaton recognising the corresponding pattern

language. As we are particularly interested in the number of counters a Janus automaton needs, we introduce an instrument to determine the quality of Janus operating modes with respect to the number of counters that are required to actually construct a Janus automaton.

Definition 4. *Let $(D_1, D_2, \ldots, D_k)$ be a Janus operating mode for a terminal-free pattern $\alpha := y_1 \cdot y_2 \cdot \ldots \cdot y_n$. Let $D = (d'_1, d'_2, \ldots, d'_{k'})$ with $k' = \sum_{i=1}^{k} |\overline{D_i}|$ be the tuple obtained from concatenating all tuples $\overline{D_j}$, $1 \leq j \leq k$, in the order given by the Janus operating mode. For each i, $1 \leq i \leq k'$, let $s_i := |\{x \mid \exists\, j, j'$ with $1 \leq j < i < j' \leq k', y_{d'_j} = y_{d'_{j'}} = x \neq y_{d'_i}\}|$. Finally let the* counter number of $(D_1, D_2, \ldots, D_k)$ *(denoted by* $\mathrm{cn}(D_1, D_2, \ldots, D_k)$*) be* $\max\{s_i \mid 1 \leq i \leq k'\}$.

With regard to our example β, it can be easily verified that $\mathrm{cn}(\Delta_\beta) = 2$. The counter number of a Janus operating mode of a pattern α is an upper bound for the number of counters needed by a Janus automaton recognising $L_\Sigma(\alpha)$:

Theorem 1. *Let α be a terminal-free pattern and $(D_1, D_2, \ldots, D_k)$ be an arbitrary Janus operating mode for α. There exists a* $\mathrm{JFA}(\mathrm{cn}(D_1, \ldots, D_k) + 1)$ *M satisfying $L(M) = L_\Sigma(\alpha)$.*

Hence, the task of finding an optimal Janus automaton for a pattern is equivalent to finding an optimal Janus operating mode for this pattern. We shall investigate this problem in the subsequent section.

5 Patterns with Restricted Variable Distance

We now introduce a certain combinatorial property of terminal-free patterns, the so-called variable distance. The variable distance of a terminal-free pattern is the maximum number of different variables separating any two consecutive occurrences of a variable:

Definition 5. *The* variable distance *of a terminal-free pattern α is the smallest number $k \geq 0$ such that, for each $x \in \mathrm{var}(\alpha)$, every factorisation $\alpha = \beta \cdot x \cdot \gamma \cdot x \cdot \delta$ with $\beta, \gamma, \delta \in X^*$ and $|\gamma|_x = 0$ satisfies $|\mathrm{var}(\gamma)| \leq k$. We denote the variable distance of a terminal-free pattern α by* $\mathrm{vd}(\alpha)$.

Obviously, $\mathrm{vd}(\alpha) \leq \mathrm{var}(\alpha) - 1$ for all terminal-free patterns α. To illustrate the concept of the variable distance, we consider the slightly more involved pattern $\alpha := x_1 \cdot x_2 \cdot x_1 \cdot x_3 \cdot x_2 \cdot x_2 \cdot x_2 \cdot x_4 \cdot x_4 \cdot x_5 \cdot x_5 \cdot x_3$. In α, there are no variables between occurrences of variables x_4 or x_5 and one occurrence of x_2 between the two occurrences of x_1. Furthermore, the variables x_1 and x_3 occur between occurrences of x_2 and the variables x_2, x_4 and x_5 occur between the two occurrences of x_3. Thus, the variable distance of this pattern is 3.

The following vital result demonstrates the relevance of the variable distance, which is a lower bound for the counter number of Janus operating modes.

Theorem 2. *Let $(D_1, D_2, \ldots, D_k)$ be an arbitrary Janus operating mode for a terminal-free pattern α. Then* $\mathrm{cn}(D_1, \ldots, D_k) \geq \mathrm{vd}(\alpha)$.

In order to define a Janus operating mode satisfying $\mathrm{cn}(D_1, \ldots, D_k) = \mathrm{vd}(\alpha)$, we now consider a particular matching order:

Definition 6. *Let $\alpha := y_1 \cdot y_2 \cdot \ldots \cdot y_n$ be a terminal-free pattern with $p := |\operatorname{var}(\alpha)|$. For each $x_i \in \operatorname{var}(\alpha)$, let $varpos_i(\alpha) := \{j_{i,1}, j_{i,2}, \ldots, j_{i,n_i}\}$ with $n_i := |\alpha|_{x_i}$, $j_{i,l} < j_{i,l+1}$, $1 \leq l \leq n_i - 1$. Let $(m_1, m_2, \ldots, m_k)$, $k = \sum_{i=1}^{p} n_i - 1$, be an enumeration of the set $\{(j_{i,l}, j_{i,l+1}) \mid 1 \leq i \leq p, 1 \leq l \leq n_i - 1\}$ such that, for every i', $1 \leq i' < k$, the left element of the pair $m_{i'}$ is smaller than the left element of $m_{i'+1}$. We call $(m_1, m_2, \ldots, m_k)$ the* canonical matching order for α.

Proposition 1. *Let α be a terminal-free pattern. The canonical matching order for α is a complete matching order.*

For instance, the canonical matching order for the example pattern β introduced in Section 4 is $((1,3),(2,4),(4,6),(5,7))$. We proceed with the definition of a Janus operating mode that is derived from the canonical matching order. It is vital for the correctness of our results, that we first move the left head and then the right head. This is easily possible if for two consecutive matching positions $(l', r'), (l, r)$, $l < r'$. If this condition is not satisfied, then the left head may pass the right one, which conflicts with the definition of Janus operating modes. Therefore, in this case, we move the left head and right head alternately.

Definition 7. *Let $(m_1, m_2, \ldots, m_k)$ be the canonical matching order for a terminal-free pattern α. For any $m_i := (j_1, j_2)$ and $m_{i-1} := (j'_1, j'_2)$, $2 \leq i \leq k$, let $(p_1, p_2, \ldots, p_{k_1}) := g(j'_1, j_1)$ and $(p'_1, p'_2, \ldots, p'_{k_2}) := g(j'_2, j_2)$, where g is the function introduced in Definition 3. If $j_1 \leq j'_2$, then we define*

$$D_i := ((p_1,\lambda),(p_2,\lambda),\ldots,(p_{k_1},\lambda),(p'_1,\rho),(p'_2,\rho),\ldots,(p'_{k_2},\rho),(j_2,\rho),(j_1,\lambda)) \ .$$

If, on the other hand, $j'_2 < j_1$, we define D_i in three parts

$$\begin{aligned} D_i := ((p_1,\lambda),(p_2,\lambda),\ldots,(j'_2,\lambda),& \\ (j'_2+1,\rho),(j'_2+1,\lambda),(j'_2+2,\rho),(j'_2+2,\lambda),\ldots,(j_1-1,\rho),(j_1-1,\lambda),& \\ (j_1,\rho),(j_1+1,\rho),\ldots,(j_2-1,\rho),(j_2,\rho),(j_1,\lambda)) \ . & \end{aligned}$$

Finally, $D_1 := ((1,\rho),(2,\rho),\ldots,(j-1,\rho),(j,\rho),(1,\lambda))$, where $m_1 = (1, j)$. The tuple $(D_1, D_2, \ldots, D_k)$ is called the canonical Janus operating mode.

If we derive a Janus operating mode from the canonical matching order for β as described in Definition 7 we obtain the canonical Janus operating mode $(((1,\rho),(2,\rho),(3,\rho),(1,\lambda)),((4,\rho),(2,\lambda)),((3,\lambda),(5,\rho),(6,\rho),(4,\lambda)),((7,\rho),(5,\lambda)))$. This canonical Janus operating mode has a counter number of 1, so its counter number is smaller than the counter number of the example Janus operating mode Δ_β given in Section 4 and, furthermore, equals the variable distance of β. With Theorem 2 we conclude that the canonical Janus operating mode for β is optimal. The next lemma shows that this holds for every pattern and, together with Theorem 1, we deduce our first main result, namely that for arbitrary patterns α, there exists a $\mathrm{JFA}(\mathrm{vd}(\alpha)+1)$ exactly accepting $L_\Sigma(\alpha)$.

Lemma 2. *Let α be a terminal-free pattern and let $(D_1, D_2, \ldots, D_k)$ be the canonical Janus operating mode for α. Then* $\mathrm{cn}(D_1, \ldots, D_k) = \mathrm{vd}(\alpha)$.

Theorem 3. *Let α be a terminal-free pattern. There exists a* $\mathrm{JFA}(\mathrm{vd}(\alpha) + 1)$ *M such that $L(M) = L_\Sigma(\alpha)$.*

The Janus automaton obtained from the canonical Janus operating mode for a pattern α is called the *canonical Janus automaton*. Theorem 3 shows the optimality of the canonical automaton. However, this optimality is subject to a vital assumption: we assume that the automaton needs to know the length of a factor in order to move an input head over this factor.

As stated above, the variable distance is the crucial parameter when constructing canonical Janus automata for pattern languages. We obtain a polynomial time match test for any class of patterns with a restricted variable distance:

Theorem 4. *There is a computable function that, given any terminal-free pattern α and $w \in \Sigma^*$, decides on whether $w \in L_\Sigma(\alpha)$ in time* $\mathrm{O}(|\alpha|^3\, |w|^{(\mathrm{vd}(\alpha)+4)})$.

As mentioned in the introduction, this main result also holds for more general classes of extended regular expressions. We anticipate, though, that the necessary amendments to our definitions involve some technical hassle.

References

1. Aho, A.: Algorithms for finding patterns in strings. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. Algorithms and Complexity, vol. A, pp. 255–300. MIT Press, Cambridge (1990)
2. Angluin, D.: Finding patterns common to a set of strings. Journal of Computer and System Sciences 21, 46–62 (1980)
3. Câmpeanu, C., Salomaa, K., Yu, S.: A formal study of practical regular expressions. International Journal of Foundations of Computer Science 14, 1007–1018 (2003)
4. Ehrenfeucht, A., Rozenberg, G.: Finding a homomorphism between two words is NP-complete. Information Processing Letters 9, 86–88 (1979)
5. Friedl, J.E.F.: Mastering Regular Expressions, 3rd edn. O'Reilly, Sebastopol (2006)
6. Guy, R.K.: The money changing problem. In: Unsolved Problems in Number Theory, 3rd edn., ch. C7, pp. 171–173. Springer, New York (2004)
7. Ibarra, O.: On two-way multihead automata. Journal of Computer and System Sciences 7, 28–36 (1973)
8. Ibarra, O.: Reversal-bounded multicounter machines and their decision problems. Journal of the ACM 25, 116–133 (1978)
9. Ibarra, O., Pong, T.-C., Sohn, S.: A note on parsing pattern languages. Pattern Recognition Letters 16, 179–182 (1995)
10. Jiang, T., Kinber, E., Salomaa, A., Salomaa, K., Yu, S.: Pattern languages with and without erasing. International Journal of Computer Mathematics 50, 147–163 (1994)
11. Shinohara, T.: Polynomial time inference of extended regular pattern languages. In: Goto, E., Furukawa, K., Nakajima, R., Nakata, I., Yonezawa, A. (eds.) RIMS 1982. LNCS, vol. 147, pp. 115–127. Springer, Heidelberg (1983)
12. Shinohara, T.: Polynomial time inference of pattern languages and its application. In: Proc. 7th IBM MFCS, pp. 191–209 (1982)

A Challenging Family of Automata for Classical Minimization Algorithms*

Giusi Castiglione[1], Cyril Nicaud[2], and Marinella Sciortino[1]

[1] DMI, Università di Palermo, via Archirafi, 34 - 90123 Palermo, Italy
{giusi,mari}@math.unipa.it
[2] LIGM, Université Paris Est, 77454 Marne-la-Vallee Cedex 2, France
nicaud@univ-mlv.fr

Abstract. In this paper a particular family of deterministic automata that was built to reach the worst case complexity of Hopcroft's state minimization algorithm is considered. This family is also challenging for the two other classical minimization algorithms: it achieves the worst case for Moore's algorithm, as a consequence of a result by Berstel et al., and is of at least quadratic complexity for Brzozowski's solution, which is our main contribution. It therefore constitutes an interesting family, which can be useful to measure the efficiency of implementations of well-known or new minimization algorithms.

1 Introduction

Regular languages are possibly infinite sets of words that can be finitely represented in many ways, as stated by Kleene's theorem, such as finite semigroups, regular expressions, finite state automata, etc. Amongst them, deterministic finite state automata (DFA) are of particular interest, since they describe simple and deterministic machines that characterize regular languages, often leading to efficient algorithms for basic tests such as membership problem, testing emptiness, and so on.

Amongst all DFA's recognizing a regular language L only one is of minimal size and is called the *minimal automaton* of L. Representing a regular language through its minimal automaton is an advantageous strategy in many applications including compilers and text searching: it is both a compact and efficient way to encode a regular language, and the uniqueness can be used to test whether two regular languages are equal.

Some of the best known and most popular algorithmic strategies for computing the minimal automaton of a language given by an automaton are due to Moore [14], Hopcroft [13] and Brzozowski [7]. Moore's and Hopcroft's algorithms operate on DFA's and use successive refinements of a partition of the set of states. Starting from an automaton with n states over an alphabet of size k, Moore's algorithm computes the minimal automaton in time $\Theta(kn^2)$. The

* Partially supported by MIUR Project *Mathematical aspects and emerging applications of automata and formal languages.*

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 251–260, 2011.

running time of Hopcroft's algorithm is $\Theta(kn \log n)$, which represents the fastest known solution to the automata minimization problem in the worst case. Brzozowski's method [7] operates by reversal and determinization repeated twice and it can be also applied to a non-deterministic finite state automaton. A detailed description of the algorithm can be found in [10]. The complexity is exponential in the worst case, though it has proven to be efficient in some particular situations [1], and is empirically better than other solutions when starting with a non-deterministic finite state automaton [16].

The importance of minimization algorithms has motivated many works in various ways. One one hand, new efficient algorithms for specific families of automata have been developed: for instance for acyclic automata [15,11], local automata [4], or Aho-Corasick automata [1]. On the other hand, efforts have been made to understand the three basic algorithms better. Several articles deal with the worst cases of Hopcroft's algorithms [5,9], the average complexity of Moore's algorithm has been analyzed [3] and various experimentations have been realized [2] to compare the performance of minimization algorithms on particular classes of automata and to construct a taxonomy [17].

In [8,9] two authors of this article, together with Antonio Restivo, exhibited a family of DFA's that reaches the worst case of Hopcroft's algorithm for a unary or a binary alphabet. In both cases, the automata are constructed starting from the well-known finite Fibonacci words and the time complexity is computed by using some combinatorial properties of such words.

This raises the natural question of the complexity of minimizing automata of this family using the two other algorithms. Answering this question is the purpose of this paper, and we show that this family, built for Hopcroft's solution, is challenging for Moore's algorithm and Brzozowski's algorithms too, being in $\Omega(n^2)$ for both. In particular, the time complexity of Moore's algorithm is here easily deduced from the literature, the result for Brzozowski's algorithm is the original contribution of the paper. Considering other well-known algorithms as an alternative strategy when Hopcroft's algorithm reaches its worst case is therefore not a good solution in this case.

The paper is organized as follows. In the first section we give some basic definitions and results about automata minimization. We define the Nerode equivalence that defines the minimal automaton equivalent to a given DFA. We describe how this equivalence relation is computed by Moore's algorithm and the time complexity of this algorithm with respect to the number of states of the initial DFA. Then, in the Section 3 we describe Brzozowski's algorithm. In Section 4 we introduce some combinatorial properties of Fibonacci words that are used in the computation of the time complexity of Brzozowski's algorithm on the word automata defined in Section 5.

2 Preliminaries on Automata and Moore's Minimization

A finite state automaton is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, Σ is a finite alphabet, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states and δ is the transition function from $Q \times \Sigma$ to 2^Q.

The finite state automaton is *deterministic* iff it has a unique initial state and δ is a mapping from $Q \times \Sigma$ to Q. By (p, x, q), with $x \in \Sigma$, we denote the transition from the state p to $q = \delta(p, x)$. The automaton $\mathcal{A}$ is *complete* if the map δ is defined for each pair $(p, x) \in Q \times \Sigma$. A *path* of $\mathcal{A}$ labeled by the word $v = x_1 x_2 ... x_n \in \Sigma^*$ is a sequence (q_i, x_i, q_{i+1}), $i = 1, ..., n$ of consecutive transitions. In such a case we write $\delta(q_1, v) = q_{n+1}$. In the case $q_1 \in I$ and $q_{n+1} \in F$ we say that the word v is *recognized* by $\mathcal{A}$. The language $\mathcal{L}(\mathcal{A})$ *recognized* by $\mathcal{A}$ is the set of all words recognized by $\mathcal{A}$.

An automaton is *minimal* if it has the minimum number of states among all its equivalent deterministic finite state automata (i.e. recognizing the same language). For each regular language there exists a unique minimal DFA. It can be computed using the Nerode equivalence. Given a state $p \in Q$, we define the language

$$\mathcal{L}_p(\mathcal{A}) = \{v \in \Sigma^* \mid \delta(p, v) \in F\}.$$

The *Nerode equivalence* on Q, denoted by $\sim$, is defined as follows: for $p, q \in Q$, $p \sim q$ if $\mathcal{L}_p(\mathcal{A}) = \mathcal{L}_q(\mathcal{A})$. It is known that $\sim$ is a congruence of $\mathcal{A}$, i.e. for any $a \in \Sigma$, $p \sim q$ implies $\delta(p, a) \sim \delta(q, a)$. It is also known that the Nerode equivalence is the coarsest congruence of $\mathcal{A}$ that saturates F, i.e. such that F is a union of classes of the congruence.

The Nerode equivalence is, effectively, computed by the Moore construction. For any integer $k \geq 0$, the Moore equivalence $\sim_k$ is defined the following way:

$$\mathcal{L}_p^k(\mathcal{A}) = \{v \in \mathcal{L}_p(\mathcal{A}) \mid |v| \leq k\}; \qquad p \sim_k q \Leftrightarrow \mathcal{L}_p^k(\mathcal{A}) = \mathcal{L}_q^k(\mathcal{A}),\ \forall p, q \in Q.$$

The *depth* of the finite automaton $\mathcal{A}$ is the smallest k such that the Moore equivalence $\sim_k$ equals the Nerode equivalence $\sim$. It is also the smallest k such that $\sim_k$ equals $\sim_{k+1}$.

Theorem 1 (Moore). *The depth of the deterministic finite state automaton* $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ *is at most* $|Q| - 2$.

Let $\mathcal{P} = \{Q_1, Q_2, ..., Q_m\}$ be the partition corresponding to the Nerode equivalence. For $q \in Q_i$, the class Q_i is denoted by $[q]$. Then the minimal automaton that recognizes $\mathcal{L}(\mathcal{A})$ is $\mathcal{MA} = (Q_M, \Sigma, \delta_M, q_{0_M}, F_M)$, where: $Q_M = \{Q_1, Q_2, ..., Q_m\}$, $q_{0_M} = [q_0]$, $\delta_M([q], a) = [\delta(q, a)]$, $\forall\ q \in Q$, $\forall\ a \in \Sigma$ and $F_M = \{[q] \mid q \in F\}$.

Moore's minimization algorithm computes the minimal automaton and it is described in Figure 1. It starts from the partition $\mathcal{P} = \{F, F^c\}$ (where F^c denotes the complement of F) which corresponds to the equivalence $\sim_0$. For each $a \in \Sigma$ we denote by $a^{-1}\mathcal{P}$ the partition in which each class the inverse image of δ (with respect to a) of a classe of $\mathcal{P}$. Then, at each iteration, the partition corresponding to the equivalence $\sim_{i+1}$ is computed from the one corresponding to the equivalence $\sim_i$, using the fact that $p \sim_{k+1} q$ iff $p \sim_k q$ and for all $a \in \Sigma$, $\delta(p, a) \sim_k \delta(q, a)$. The algorithm halts when no new partition refinement is obtained, and the result is the Nerode equivalence. Each iteration is performed in time $\Theta(|Q|)$ using a radix sort. The time complexity of Moore' s algorithm applied to $\mathcal{A}$ is therefore $\Theta(d\,|Q|)$, where d is the depth of $\mathcal{A}$.

Algorithm 1: Moore's algorithm on $(Q, \Sigma, \delta, q_0, F)$

$\mathcal{P} = \{F, F^c\}$
repeat
  $\mathcal{W} \leftarrow \mathcal{P}$
  for $a \in \Sigma$ **do**
    $\mathcal{P}_a \leftarrow a^{-1}\mathcal{P}$
  $\mathcal{P} \leftarrow \mathcal{P} \cap \bigcap_{a \in \Sigma} \mathcal{P}_a$
until $\mathcal{W} = \mathcal{P}$
return $(Q_M, \Sigma, \delta_M, q_{0_M}, F_M)$

Fig. 1. Moore's algorithm that computes the minimal DFA equivalent to a given deterministic finite state automaton

3 Subset Construction and Brzozowski's Algorithm

In this section we study the time complexity of Brzozowski's minimization. First, we give some preliminary definitions and notations.

Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a finite state automaton. We say that a state of Q is *accessible* (resp. *coaccessible*) if there exists a path from an initial state to this state (resp. from this state to a final state). By $d(\mathcal{A})$ we denote the deterministic finite state automaton equivalent to $\mathcal{A}$, $d(\mathcal{A}) = (Q_d, \Sigma, \delta_d, \mathfrak{q}_0, F_d)$, where:

- Q_d is the set of subsets $\mathfrak{p}$ of Q
- $\mathfrak{q}_0 = I$
- $\delta_d(\mathfrak{p}, a) = \{\delta(p, a) \mid p \in \mathfrak{p}\}, \forall \mathfrak{p} \in Q_d, a \in \Sigma$
- $F_d = \{\mathfrak{p} \mid \mathfrak{p} \cap F \neq \emptyset\}$.

We call a *singleton* state any $\mathfrak{p} \in Q$ such that $|\mathfrak{p}| = 1$. The mechanism of building $d(\mathcal{A})$ (resp. the accessible part of $d(\mathcal{A})$) from $\mathcal{A}$ is called the *subset construction* (resp. *the accessible subset construction*).

The *reverse* of the automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is the automaton $r(\mathcal{A}) = (Q_r, \Sigma, \delta_r, I_r, F_r) = (Q, \Sigma, \delta_r, F, I)$, where for each $a \in \Sigma$ and $q \in Q_r$, $\delta_r(q, a) = \{p \in Q \mid q \in \delta(p, a)\}$. Observe that when we consider the reverse $r(\mathcal{A})$ of a deterministic and complete automaton $\mathcal{A}$, for every state p and every letter x in the alphabet, there is exactly one transition labeled by x ending in p in $r(\mathcal{A})$. Hence, for every state $\mathfrak{p}$ of $d(r(\mathcal{A}))$, either $p \in \delta_r(\mathfrak{p}, x)$ or $p \in \delta_r(\mathfrak{p}^c, x)$. The following proposition is a consequence of this argument.

Proposition 1. *Let $\mathcal{A}$ be a deterministic and complete finite state automaton. For every $x \in \Sigma$, $(\mathfrak{p}, x, \mathfrak{q})$ is a transition in $d(r(\mathcal{A}))$ if and only if $(\mathfrak{p}^c, x, \mathfrak{q}^c)$ is a transition in $d(r(\mathcal{A}))$.*

Brzozowski's algorithm is based on the fact that if $\mathcal{A}$ is a codeterministic and coaccessible finite automaton recognizing L, then the accessible part of $d(\mathcal{A})$ is the minimal automaton of L [7]. Given an automaton $\mathcal{A}$, let $\mathcal{B}$ be the accessible part of $r(\mathcal{A})$, the reverse of $\mathcal{A}$; the automaton $r(\mathcal{B})$ recognizes the same language

Algorithm 2: Accessible Subset Construction of $(Q, \Sigma, \delta, I, F)$

```
1  S = Q_d = {I}
2  F_d = ∅
3  while S ≠ ∅ do
4      𝕡 ← Extract from S
5      if 𝕡 ∩ F ≠ ∅ then Add 𝕡 in F_d
6      for a ∈ Σ do
7          𝕢 = ∅
8          for p ∈ 𝕡 do 𝕢 ← 𝕢 ∪ δ(p, a)
9          if 𝕢 ∉ Q_d then
10             Add 𝕢 in Q_d
11             Add 𝕢 in S
12         δ_d(𝕡, a) ← 𝕢
13 return (Q_d, Σ, δ_d, {I}, F_d)
```

Fig. 2. Algorithm that computes the accessible part of the subset construction of an automaton

as $\mathcal{A}$ and by construction, it is both codeterministic and coaccessible. Hence the accessible part of $r(\mathcal{B})$ is the minimal automaton of $\mathcal{A}$.

The algorithm therefore consists of computing two reversals and performing two subset constructions. Since the reverse of an automaton can easily be computed in linear time with respect to its size (number of states and transitions), the critical part is the subset construction, or more precisely the accessible subset construction. The accessible subset construction can be performed as described in Fig. 2. In the process, all accessible states of $d(\mathcal{A})$ are extracted exactly once from S. Moreover, considering Line 8 only, at least $|\mathbb{p}|$ iterations are necessary to compute $\delta_d(\mathbb{p}, a)$. Hence, $\sum_{\mathbb{p}} |\mathbb{p}|$, where $\mathbb{p}$ ranges over all accessible states, is a lower bound of the time complexity of the accessible subset construction.

Note that different data structures can be considered for S and for Q_d, this is important especially for Lines 9, 10 and 12. We do not develop this discussion further here since the stated lower bound is enough for our purpose.

Since the reverse operation can be applied also to non-deterministic finite state automata this algorithm is able to minimize both deterministic and non-deterministic finite state automata. Because of the determinization the worst-case running time complexity of the algorithm is exponential. However in [16,2] it is experimentally verified that in practice Brzozowski's algorithm has a good performance and usually outperforms the other algorithms when applied on non-deterministic finite state automata.

4 Circular Factors of Fibonacci Words

In this section we present some combinatorial properties of finite Fibonacci words in order to show their connections with the computation of the running time of

Brzozowski's algorithm on a particular class of automata defined in next section. In particular, the following results are useful to compute the size of subsets of states in the determinization of word automata.

Let A be a finite alphabet and v, u be two words in A^*. We say that v and u are conjugate if for some words $z, w \in A^*$ one has that $v = zw$ and $u = wz$. It is easy to see that conjugation is an equivalence relation. Note that many combinatorial properties of words in A^* can be thought as properties of the respective conjugacy classes.

We say that a word $v \in A^*$ is a *circular factor* of a word w if v is a factor of some conjugate of w. Equivalently, a circular factor of w is a factor of ww of length not greater than $|w|$. Note that, while each factor of w is also a circular factor of w, there exist circular factors of a word w that are not factors of w. For instance, *ca* is a circular factor of *abc* without being factor of *abc*.

In this paper we take into consideration finite Fibonacci words and their circular factors. We denote by f_n the n-th finite Fibonacci word and $F_n = |f_n|$. Fibonacci words are defined by the recurrence relation $f_n = f_{n-1}f_{n-2}$, with $f_0 = b$, $f_1 = a$. Note that if $n > 1$ is odd (resp. even) then f_n ends by ba (resp. ab). We now state some combinatorial properties of the finite Fibonacci words used in next section.

Proposition 2. *Let f_n be the n-th Fibonacci word, with $n \geq 3$. If n is odd then the circular factor af_{n-1} has a unique occurrence, at position F_n. If n is even then the circular factor af_{n-2} has a unique occurrence, at position F_{n-1}.*

Proof. We prove the statement by induction on n. For $n = 3$ and $n = 4$ it is trivial. Let us consider an odd n, since $f_n = f_{n-1}f_{n-2}$ we have that the circular factor af_{n-1} trivially occurs in f_n at position F_n. Suppose that af_{n-1} occurs in another position $i < F_n$. Note that $af_{n-1} = af_{n-3}uaf_{n-3}$, with $u \in A^*$ and $|u| = |f_{n-4}| - 1$. If $1 \leq i \leq F_{n-1}$ we consider the factorization $f_nf_n = f_{n-1}f_{n-1}f_{n-4}f_{n-3}f_{n-2}$. We conclude that af_{n-3} has two different occurrences in $f_{n-1}f_{n-1}$ i.e. two different circular occurrences in f_{n-1}. This fact contradicts the inductive hypothesis. In the case $F_{n-1} < i < F_n$ we consider the factorization $f_nf_n = f_{n-1}f_{n-2}f_{n-2}f_{n-3}f_{n-2}$, and af_{n-3} occurs in $f_{n-2}f_{n-2}$ in a position that is not F_{n-2}, which also contradicts the hypothesis. □

Proposition 3. *Let f_n be the n-th Fibonacci word, with $n \geq 2$, and let u be a suffix of f_n that is also a prefix of f_n. Then u is equal to f_{n-2i}, for some $i > 0$.*

Proof. It follows by definition of f_n that f_{n-2i} is both a prefix and a suffix of f_n, for $i > 0$. Suppose that u is different from f_{n-2i} for all $i > 0$. Without loss of generality we can suppose that $F_{n-2} < |u| < F_{n-1}$. Let v be the prefix of f_n of length $F_n - 2$. If u is a prefix and a suffix, then it is easy to see that $F_n - |u|$ is a period of f_n and v. Note that $F_{n-2} < F_n - |u| < F_{n-1}$. So, since F_{n-2} and F_{n-1} are coprime and $F_{n-1}/F_{n-2} < 2$, it follows that F_{n-2} and $F_n - |u|$ are coprime. It is known that (cf. [12]) if F_{n-2} and F_{n-1} are both period of v, then F_{n-2} and $F_n - |u|$ are two coprime periods of v and $|v| = F_n - 2 > F_n - |u| + F_{n-2} - 2$. So, by Fine and Wilf's theorem, v should be a power of a letter. □

5 The Two Algorithms on Word Automata

In [9] the behaviour of Hopcroft's algorithm on binary automata associated to finite sturmian words has been analyzed. One of the results states that when such automata are associated to Fibonacci words Hopcroft's algorithm runs in time $\Theta(n \log n)$, where n is the size of the automaton. Here we focus our attention on the same family of automata.

Let $w = a_1 a_2 ... a_n$ be a word of length n over the binary alphabet $A = \{a, b\}$. The *word automaton associated to* w, denoted by $\mathcal{A}_w$, is the DFA $(Q, A, \delta, 1, F)$ such that $Q = \{1, 2, \cdots, n\}$, $F = \{i \in Q \mid a_i = b\}$, and with, for every $i \in Q$ and every $x \in A$,

$$\delta(i, x) = \begin{cases} i+1 & \text{if } i \neq n \text{ and } x = a_i \\ 1 & \text{otherwise} \end{cases}$$

Note that we choose to consider a binary alphabet but such definitions and results hold for automata over an alphabet with a generic size. The automata $\mathcal{A}_{f_5}$ and $r(\mathcal{A}_{f_5})$ are depicted in Fig. 3.

When w is a finite Fibonacci word, both Moore's algorithm and Brzozowski's algorithm run in time $\Omega(|Q|^2)$ on $\mathcal{A}_w$. In regards to Moore's algorithm, this fact can be easily deduced by some propositions proved in [6]. An automaton with $|Q|$ states is *slow* iff each Moore equivalence $\sim_h$, for $h \leq |Q|-2$, has $h+2$ classes. By using results in [6,9] one can infer that word automata associated to finite Fibonacci words are slow. From these results one can directly conclude that the number of steps of Moore's algorithm, in this case, is exactly $|Q| - 1$, i.e. the depth is $|Q| - 1$. Then one can easily deduce the following theorem.

Theorem 2. *Let $c_M(F_n)$ be the time complexity of Moore's algorithm on the word automaton, with F_n states, $\mathcal{A}_{f_n}$ corresponding to the n-th Fibonacci word f_n. Then*

$$c_M(F_n) = \Theta(F_n^2).$$

The analysis of Brzozowski's algorithm on word automata is much less intuitive, and it will be the goal of the rest of this section. In order to compute a lower bound for this algorithm we need to make explicit some properties of the paths in $r(\mathcal{A}_{f_n})$ closely related with the properties of some factors of Fibonacci words.

Remark 1. If $u = vx$, with $x \in A$, is a circular factor of the word f_n then there exists a path in $r(\mathcal{A}_{f_n})$ labeled by v^r. In particular, if $u = vb$ then there exists in $r(\mathcal{A}_{f_n})$ starting from an initial state and labeled with v^r.

Lemma 1. *Let f_n be the n-th fibonacci word with odd n (resp. even). A word $w = x_1 x_2 \dots x_k$ is a circular factor of f_n that occurs at position i if and only if there exists a path $(p_1, x_k, p_2)(p_2, x_{k-1}, p_3) \cdots (p_k, x_1, i)$ in $r(\mathcal{A}_{f_n})$ such that either $p_j \neq 1, \forall 1 \leq j \leq k$ or, if $p_j = 1$ for some j then $x_j x_{j+1} = ab$ (resp. $x_j x_j + 1 = ba$).*

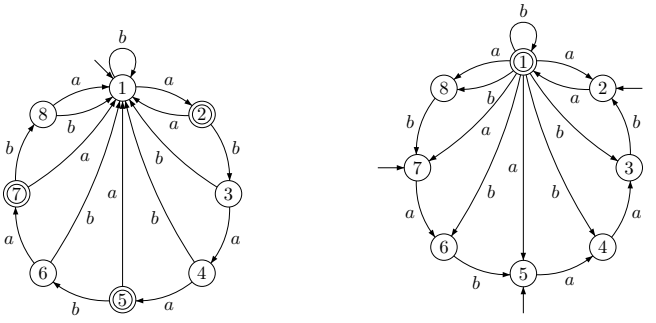


Fig. 3. The word automaton $\mathcal{A}_{f_5}$ associated to the word $f_5 = abaababa$ and $\mathrm{r}(\mathcal{A}_{f_5})$

At this point of the paper it is clear that one can find in $d(r(\mathcal{A}_{f_n}))$ some paths closely related with the properties of the circular factors that label them. In particular, occurrences of some factors determine the cardinality of subsets reached by the corresponding paths. This connection between paths in $d(r(\mathcal{A}_{f_n}))$ and factors in f_n is highlighted by the fact that proofs of the following results use properties proved in Section 4.

Theorem 3. *Let $\mathcal{A}_{f_n}$ be the word automaton associated to the n-th Fibonacci word f_n. If n is odd, then in the automaton $d(r(\mathcal{A}_{f_n}))$ the singleton state $\{F_n - 1\}$ is accessible. If n is even, then in the automaton $d(r(\mathcal{A}_{f_n}))$ the singleton state $\{F_{n-1} - 1\}$ is accessible.*

Proof. Let us suppose that n is odd. Since, by definition of Fibonacci words, $f_n = f_{n-1}f_{n-2}$ and f_{n-2} ends by ba then by Proposition 2 the word baf_{n-1} is a circular factor of f_n having a unique occurrence at position $F_n - 1$. Let us observe that $(baf_{n-1})^r = baf_{n-1}$. It follows by Lemma 1 that there exists a path in $r(\mathcal{A}_{f_n})$ labeled by af_{n-1} starting from the initial state $F_n - 1$ and passing through the state 1. Let $\mathbb{p} \in Q_d$ the state reached by such a path. From the fact described above, it follows that $F_n - 1 \in \mathbb{p}$. In order to prove the thesis we prove that $\mathbb{p} = \{F_n - 1\}$. Let us suppose that there exists another path P in $r(\mathcal{A}_{f_n})$ labeled by af_{n-1} starting from an initial state. If it does not pass through the state 1 then it ends at the state $i \neq F_n - 1$. By Lemma 1 it follows that af_{n-1} occurs in f_n at position $i \neq F_n - 1$. This fact contradicts Proposition 2. If the path P passes through the state 1 we can write $P = (p_1, x_1, p_2)(p_2, x_2, p_3) \cdots (p_r, x_r, 1)(1, x_{r+1}, p_{r+2}) \cdots (p_{F_{n-1}+1}, x_{F_{n-1}+1}, p_{F_{n-1}+2})$, where $af_{n-1} = x_1x_2 \cdots x_rx_{r+1}x_{F_{n-1}+1}$ and $p_i \neq 1$ for all $1 \leq i \leq r$. Hence we have that $x_1 \ldots x_r$ is a prefix of f_{n-1} that is a prefix of f_n, i.e. $x_1 \ldots x_r$ is both a prefix and a suffix of f_{n-1}. By using Proposition 3, $x_1 \ldots x_r$ is the palindromic prefix of a Fibonacci word f_k of length $F_k - 2$ ending by ab, then $x_{r+1}x_{r+2} = ab$. By Lemma 1 and by uniqueness of occurrences of af_{n-1} proved in Proposition 2, this path corresponds to a circular occurrence of af_{n-1} in f_n, i.e. the considered path ends at $F_n - 1$. The even case can be analogously proved. □

Corollary 1. *If n is odd (resp. even) then in the automaton $d(r(\mathcal{A}_{f_n}))$ the singleton state $\{F_n\}$ is not accessible (resp. the singleton states $\{k\}$, with $F_{n-1} \leq k \leq F_n$ are not accessible) and the singleton states $\{k\}$, with $1 \leq k \leq F_n - 1$ (resp. with $1 \leq k \leq F_{n-1} - 1$) are accessible.*

Note that since $\mathcal{A}_{f_n}$ is a complete DFA, Proposition 1 holds. We use this fact in the sequel, together with the following theorem, in order to estimate the accessible part of the automaton $d(r(\mathcal{A}_{f_n}))$.

Theorem 4. *Let $\mathcal{A}_{f_n}$ be the word automaton corresponding to the n-th Fibonacci word f_n. If n is odd, then in the automaton $d(r(\mathcal{A}_{f_n}))$ the state $\{F_n - 1\}^c$ is accessible. If n is even, then in the automaton $d(r(\mathcal{A}_{f_n}))$ the state $\{F_{n-1} - 1\}^c$ is accessible.*

The main ingredient of the proof, that we do not report for brevity, is the fact that the complement of the set of the initial states is accessible in case of n odd. In the even case we can prove that the state $\{F_{n-1} - 1\}^c$ is accessible by using the path labeled by f_{n-2}.

From Proposition 1 and Theorem 4 one can deduce the following corollaries.

Corollary 2. *If n is odd then in the automaton $d(r(\mathcal{A}_{f_n}))$ the states $\{k\}^c$, with $1 \leq k \leq F_n - 1$, are accessible.*

Corollary 3. *If n is even then in the automaton $d(r(\mathcal{A}_{f_n}))$ the states $\{k\}^c$, with $1 \leq k \leq F_{n-1} - 1$, are accessible.*

Theorem 5. *Let $c_B(F_n)$ be the time complexity of Brzozowski's algorithm on the word automaton, with F_n states, $\mathcal{A}_{f_n}$ corresponding to the n-th Fibonacci word f_n. Then*

$$c_B(F_n) = \Omega(F_n^2).$$

Proof. By the previous theorem and corollaries it follows that Q_d of $d(r(\mathcal{A}_{f_n}))$, with odd n, contains $F_n - 1$ singletons and then $F_n - 1$ subsets of cardinality $F_n - 1$. In the same way for even n, it contains $F_{n-1} - 1$ singletons and then $F_{n-1} - 1$ subsets of cardinality $F_{n-1} - 1$. Hence the thesis. □

6 Further Work

Recall that finite Fibonacci words are particular finite sturmian words. We believe that the techniques used in the proofs can be applied also to word automata defined by sturmian words. In [5] the authors define a family of unary cyclic automata associated to words and characterize sturmian words for which the associated automata represent the worst-case of Hopcroft's algorithm. Fibonacci word, for instance, is one of those. Hence, it would be interesting to characterize those sturmian words for which Brzozowski's algorithm on the associated word automata is at least quadratic. Furthermore, we want to analyze the question whether there are minimization algorithms that work better on the automata representing the extremal cases for Hopcroft's algorithm.

References

1. AitMous, O., Bassino, F., Nicaud, C.: Building the minimal automaton of A*X in linear time, when X is of bounded cardinality. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 275–287. Springer, Heidelberg (2010)
2. Almeida, M., Moreira, N., Reis, R.: On the performance of automata minimization algorithms. Technical Report DCC-2007-03, Universidade do Porto (2007)
3. Bassino, F., David, J., Nicaud, C.: On the average complexity of Moore's state minimization algorithm. In: STACS, pp. 123–134 (2009)
4. Béal, M.-P., Crochemore, M.: Minimizing local automata. In: IEEE International Symposium on Information Theory (ISIT 2007), pp. 1376–1380 (2007)
5. Berstel, J., Boasson, L., Carton, O.: Continuant polynomials and worst-case behavior of Hopcroft's minimization algorithm. TCS 410, 2811–2822 (2009)
6. Berstel, J., Boasson, L., Carton, O., Fagnot, I.: Sturmian trees. Theory of Computing Systems 46(3), 443–478 (2010)
7. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. Mathematical Theory of Automata 12, 529–561 (1962)
8. Castiglione, G., Restivo, A., Sciortino, M.: Circular sturmian words and Hopcroft's algorithm. TCS 410, 4372–4381 (2009)
9. Castiglione, G., Restivo, A., Sciortino, M.: On extremal cases of Hopcroft's algorithm. TCS 411(38-39), 3414–3422 (2010)
10. Champarnaud, J.-M., Khorsi, A., Paranthoën, T.: Split and join for minimizing: Brzozowski's algorithm. In: PSC 2002, pp. 96–104 (2002)
11. Daciuk, J., Watson, R.E., Watson, B.W.: Incremental construction of acyclic finite-state automata and transducers. In: Finite State Methods in Natural Language Processing, Bilkent University, Ankara, Turkey (1998)
12. de Luca, A., Mignosi, F.: Some combinatorial properties of Sturmian words. TCS 136(2), 361–385 (1994)
13. Hopcroft, J.E.: An $n \log n$ algorithm for mimimizing the states in a finite automaton. In: Theory of Machines and Computations, Proc. Internat. Sympos. Technion, Haifa, pp. 189–196. Academic Press, New York (1971)
14. Moore, E.F.: Gedaken experiments on sequential machines, pp. 129–153. Princeton University Press, Princeton (1956)
15. Revuz, D.: Minimisation of acyclic deterministic automata in linear time. TCS 92(1), 181–189 (1992)
16. Tabakov, D., Vardi, M.: Experimental evaluation of classical automata constructions. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 396–411. Springer, Heidelberg (2005)
17. Watson, B.: A taxonomy of finite automata minimization algorithms. Technical Report 93/44, Eindhoven Univ. of Tech., Faculty of Math. and Comp. Sc. (1994)

State of Büchi Complementation*

Ming-Hsien Tsai[1], Seth Fogarty[2], Moshe Y. Vardi[2], and Yih-Kuen Tsay[1]

[1] National Taiwan University
[2] Rice University

Abstract. Büchi complementation has been studied for five decades since the formalism was introduced in 1960. Known complementation constructions can be classified into Ramsey-based, determinization-based, rank-based, and slice-based approaches. For the performance of these approaches, there have been several complexity analyses but very few experimental results. What especially lacks is a comparative experiment on all the four approaches to see how they perform in practice. In this paper, we review the state of Büchi complementation, propose several optimization heuristics, and perform comparative experimentation on the four approaches. The experimental results show that the determinization-based Safra-Piterman construction outperforms the other three and our heuristics substantially improve the Safra-Piterman construction and the slice-based construction.

1 Introduction

Büchi automata are nondeterministic finite automata on infinite words that recognize ω-regular languages. It is known that Büchi automata are closed under Boolean operations, namely union, intersection, and complementation. Complementation was first studied by Büchi in 1960 for a decision procedure for second-order logic [3]. Complementation of Büchi automata is significantly more complicated than that of nondeterministic finite automata on finite words. Given a nondeterministic finite automaton on finite words with n states, complementation yields an automaton with 2^n states through the subset construction. Indeed, for nondeterministic Büchi automata, the subset construction is insufficient for complementation. In fact, Michel showed in 1988 that blow-up of Büchi complementation is at least $n!$ (approximately $(n/e)^n$ or $(0.36n)^n$), which is much higher than 2^n [17]. This lower bound was later sharpened by Yan to $(0.76n)^n$ [31], which was matched by an upper bound by Schewe [21].

There are several applications of Büchi complementation in formal verification, for example, verifying whether a system satisfies a property by checking if the intersection of the system automaton and the complement of the property automaton is empty [27], testing the correctness of an LTL translation algorithm without a reference algorithm, etc. [9]. Although recently many works focus on

* Work supported in part by the National Science Council, Taiwan (R.O.C.) under grant NSC97-2221-E-002-074-MY3, by NSF grants CCF-0613889, ANI-0216467, CCF-0728882, and OISE-0913807, by BSF grant 9800096, and by gift from Intel.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 261–271, 2011.

universality and containment testing without going explicitly through complementation [5,6,4], it is still unavoidable in some cases [16].

Known complementation constructions can be classified into four approaches: Ramsey-based approach [3,22], determinization-based approach [20,18,2,19], rank-based approach [24,15,13], and slice-based approach [10,30]. The first three approaches were reviewed in [29]. Due to the high complexity of Büchi complementaton, optimization heuristics are critical to good performance [9,7,21,11,14]. Unlike the rich theoretical development, empirical studies of Büchi complementation have been rather few [14,9,11,26], as much recent emphasis has shifted to universality and containment. A comprehensive empirical study would allow us to evaluate the performance of these complementation approaches.

In this paper, we review the four complementation approaches and perform comparative experimentation on the best construction in each approach. Although the conventional wisdom is that the nondeterministic constructions are better than the deterministic construction, due to better worse-case bounds, the experimental results show that the deterministic construction is the best for complementation in general. At the same time, the Ramsey-based approach, which is competitive in universality and containment testing [1,5,6], performs rather poorly in our complementation experiments. We also propose optimization heuristics for the determinization-based construction, the rank-based construction, and the slice-based construction. The experiment shows that the optimization heuristics substantially improve the three constructions. Overall, our work confirms the importance of experimentation and heuristics in studying Büchi complementation, as worst-case bounds are poor guides to actual performance.

This paper is organized as follows. Some preliminaries are given in Section 2. In Section 3, we review the four complementation approaches. We discuss the results of our comparative experimentation on the four approaches in Section 4. Section 5 describes our optimization heuristics and Section 6 shows the improvement made by our heuristics. We conclude in Section 7. More results of the experiments in Section 4 and Section 6 and further technical details regarding some of the heuristics can be found in [25].

2 Preliminaries

A *nondeterministic* ω-automaton A is a tuple $(\Sigma, Q, q_0, \delta, \mathcal{F})$, where Σ is the finite alphabet, Q is the finite state set, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, and $\mathcal{F}$ is the acceptance condition, to be described subsequently. A is *deterministic* if $|\delta(q, a)| = 1$ for all $q \in Q$ and $a \in \Sigma$.

Given an ω-automaton $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ and an infinite word $w = a_0 a_1 \cdots \in \Sigma^\omega$, a *run* ρ of A on w is a sequence $q_0 q_1 \cdots \in Q^\omega$ satisfying $\forall i : q_{i+1} \in \delta(q_i, a_i)$. A run is *accepting* if it satisfies the acceptance condition. A word is *accepted* if there is an accepting run on it. The *language* of an ω-automaton A, denoted by $L(A)$, is the set of words accepted by A. An ω-automaton A is *universal* if $L(A) = \Sigma^\omega$. A state is *live* if it occurs in an accepting run on some word, and is *dead* otherwise. Dead states can be discovered using a nonemptiness algorithm, cf. [28], and can be pruned off without affecting the language of the automaton.

Let ρ be a run and $inf(\rho)$ be the set of states that occur infinitely often in ρ. Various ω-automata can be defined by assigning different acceptance conditions: *Büchi condition* where $\mathcal{F} \subseteq Q$ and ρ satisfies the condition iff $inf(\rho) \cap \mathcal{F} \neq \emptyset$; *Rabin condition* where $\mathcal{F} \subseteq 2^Q \times 2^Q$ and ρ satisfies the condition iff there exists $(E, F) \in \mathcal{F}$ such that $inf(\rho) \cap E = \emptyset$ and $inf(\rho) \cap F \neq \emptyset$; *parity condition* where $\mathcal{F} : Q \to \{0, 1, \ldots, 2r\}$ and ρ satisfies the condition iff $min\{\mathcal{F}(q) \mid q \in inf(\rho)\}$ is even. $\mathcal{F}(q)$ is called the parity of a state q.

We use a system of three-letter acronyms to denote these ω-automata. The first letter indicates whether the automaton is **n**ondeterministic or **d**eterministic. The second letter indicates whether the acceptance condition is **B**üchi, **R**abin, or **p**arity. The third letter is always a "**W**" indicating the automaton accepts words. For example, NBW stands for a nondeterministic Büchi automaton and DPW stands for a deterministic parity automaton.

Given an ω-automaton A and an infinite word w, the *run tree* of A on w is a tree where the vertices of a (full) branch form a run of A on w and there is a corresponding branch for every run of A on w. The *split tree* of A on w is a binary tree that abstracts the run tree by grouping accepting successors and nonaccepting successors of states in a vertex respectively into the left child and the right child. The *reduced split tree* of A on w is a binary tree obtained from the split tree of A on w by removing a state from a vertex if it also occurs in a vertex to the left on the same level and removing a vertex if it contains no state. An NBW accepts a word if there is a left-recurring branch in the reduced split tree. A *slice* is a sequence of state sets representing all vertices on a same level of a reduced split tree in an order from left to right.

3 Historical Review

Ramsey-based approach. The very first complementation construction introduced by Büchi in 1960 involves a Ramsey-based combinatorial argument and results in a $2^{2^{O(n)}}$ blow-up in the state size [3]. This construction was later improved by Sistla, Vardi, and Wolper to reach a single-exponential complexity $2^{O(n^2)}$ [22]. In the improved construction, referred to as `Ramsey` in this paper, the complement is obtained by composing certain automata among a set of Büchi automata which form a partition of Σ^ω, based on Ramsey's Theorem. Various optimization heuristics for the Ramsey-based approach are described in [1,6], but the focus in these works is on universality and containment. In spite of the quadratic exponent of the Ramsey-based approach, it is shown in [1,5,6] to be quite competitive for universality and containment.

Determinization-based approach. Safra's $2^{O(n \log n)}$ construction is the first complementation construction that matches the $\Omega(n!)$ lower bound [20]. Later on, Muller and Schupp introduced a similar determinization construction which records more information and yields larger complements in most cases, but can be understood more easily [18,2]. In [19], Piterman improved Safra's construction by using a more compact structure and using parity automata as the intermediate deterministic automata, which yields an upper bound of n^{2n}. Piterman's

construction, referred to as `Safra-Piterman` in this paper, performs complementation in stages: from NBW to DPW, from DPW to complement DPW, and finally from complement DPW to complement NBW. The idea is the use of (1) a compact Safra tree to capture the history of all runs on a word and (2) marks to indicate whether a run passes an accepting state again or dies.

Since the determinization-based approach performs complementation in stages, different optimization techniques can be applied separately to the different stages. For instance, several optimization heuristics on Safra's determinization and on simplifying the intermediate DRW were proposed by Klein and Baier [14].

Rank-based approach. The rank-based approach, proposed by Kupferman and Vardi, uses rank functions to measure the progress made by a node of a run tree towards fair termination [15]. The basic idea of this approach may be traced back to Klarlund's construction with a more complex measure [13]. Both constructions have complexity $2^{O(n \log n)}$. There were also several optimization techniques proposed in [9,7,11]. A final improvement was proposed recently by Schewe [21] to the construction in [7]. The later construction performs a subset construction in the first phase. In the second phase, it continually guesses ranks from some point and verifies the guesses. Schewe proposed doing this verification in a piecemeal fashion. This yields a complement with $O((0.76n)^n)$ states, which matches the known lower bound modulo an $O(n^2)$ factor. We refer to the construction with Schewe's improvement as `Rank` in this paper.

Unlike the determinization-based approach that collects information from the history, the rank-based approach guesses ranks bounded by $2(n-|\mathcal{F}|)$ and results in many nondeterministic choices. This nondeterminism means that the rank-based construction often creates more useless states because many guesses may be verified later to be incorrect.

Slice-based approach. The slice-based construction was proposed by Kähler and Wilke in 2008 [10]. The blow-up of the construction is $4(3n)^n$ while its preliminary version in [30], referred to as `Slice` here, has a $(3n)^n$ blow-up[1]. Unlike the previous two approaches that analyze run trees, the slice-based approach analyzes reduced split trees. The construction `Slice` uses slices as states of the complement and performs a construction based on the evolution of reduced split trees in the first phase. By decorating vertices in slices at some point, it guesses whether a vertex belongs to an infinite branch of a reduced split tree or the vertex has a finite number of descendants. In the second phase, it verifies the guesses and enforces that accepting states will not occur infinitely often.

The first phase of `Slice` in general creates more states than the first phase of `Rank` because of an ordering of vertices in the reduced split trees. Similar to `Rank`, `Slice` also introduces nondeterministic choices in guessing the decorations. While `Rank` guesses ranks bounded by $2(n - |\mathcal{F}|)$ and continually guesses ranks in the second phase, `Slice` guesses only once the decorations from a fixed set of size 3 at some point.

[1] The construction in [10] has a higher complexity than its preliminary version because it treats complementation and disambiguation in a uniform way.

4 Comparison of Complementation Approaches

We choose four representative constructions, namely `Ramsey`, `Safra-Piterman`, `Rank`, and `Slice`, that are considered the most efficient construction in each approach. These constructions are implemented in the GOAL tool [26]. We randomly generate 11,000 automata with an alphabet of size 2 and a state set of size 15 from combinations of 11 transition densities and 10 acceptance densities. For each automaton $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$ with a given state size n, symbol $a \in \Sigma$, transition density r, and acceptance density f, we make $t \in \delta(s, a)$ for $\lceil rn \rceil$ pairs of states $(s, t) \in Q^2$ uniformly chosen at random and add $\lceil fn \rceil$ states to $\mathcal{F}$ uniformly at random. Our parameters were chosen to generate a large set of complementation problems, ranging from easy to hard. The experiment was run in a cluster at Rice University (http://rcsg.rice.edu/sugar/int/). For each complementation task, we allocate one 2.83 GHz CPU and 1 GB memory. The timeout of a complementation task is 10 minutes.

Table 1. The results of comparing the four representative constructions

Constructions	Eff. Samples	S_R (Win)	S_L (Win)	S_L/S_R	T	M
`Ramsey`	-	-	-	-	11,000	0
`Safra-Piterman`	3,826	65.01 (2,797.0)	22.63 (1,066.17)	0.35	5	0
`Rank`		310.52 (1,025.5)	33.81 (1,998.67)	0.11	5,303	0
`Slice`		887.43 (3.5)	54.58 (761.17)	0.06	3,131	3,213

We only collect state-size information from *effective samples*, which are tasks finished successfully by all constructions. Otherwise, a construction may be considered to be worse in producing more states because it is better in finishing more tasks. The experimental results are listed in Table 1 where S_R is the average number of reachable states created in an effective sample, S_L is the average number of live states created in an effective sample, T is the total number of timed-out tasks, and M is the total number of tasks that run out-of-memory. The Win column of S_R (resp., S_L) is the share of effective samples where one construction produces smallest complements in terms of reachable states (resp., live states). `Ramsey` is not competitive at all in complementation and is separated from the other three in Table 1 because it failed to finish any task, even though it is competitive in universality and containment, as shown in [1,5,6].

The S_R, S_L, and T columns show that the `Safra-Piterman` is the best both in average state size and in running time. The low S_L/S_R ratio shows that `Rank` and `Slice` create more dead states that can be easily pruned off. The Win columns show that although `Rank` generates more dead states, it produces more complements that are the smallest after pruning dead states. `Slice` becomes much closer to `Safra-Piterman` in the Win column of S_L because more than one half of the 3,826 effective samples are universal automata. Except `Ramsey`, `Slice` has the most unfinished tasks and produces many more states than `Safra-Piterman` and `Rank`. As we show later, we can improve the performance of `Slice` significantly by employing various optimization heuristics.

5 Optimization Techniques

5.1 For `Safra-Piterman`

`Safra-Piterman` performs complementation via several intermediate stages: starting with the given NBW, it computes first an equivalent DPW, then the complement DPW, and finally the complement NBW. We address (1) the simplification of the complement DPW, which results in an NPW, and (2) the conversion from an NPW to an equivalent NBW.

Simplifying DPW by simulation. (`+S`). For the simplification of the complement DPW, we borrow from the ideas of Somenzi and Bloem [23]. The direct and reverse simulation relations they introduced are useful in removing transitions and possibly states of an NBW while retaining its language. We define the simulation relations for an NPW in order to apply the same simplification technique. Given an NPW $(\Sigma, Q, q_0, \delta, \mathcal{F})$ and two states $q_i, q_j \in Q$, q_j *directly simulates* q_i iff (1) for all $q_i' \in \delta(q_i, a)$, there is $q_j' \in \delta(q_j, a)$ such that q_j' directly simulates q_i', and (2) $\mathcal{F}(q_i) = \mathcal{F}(q_j)$. Similarly, q_j *reversely simulates* q_i iff (1) for all $q_i' \in \delta^{-1}(q_i, a)$, there is $q_j' \in \delta^{-1}(q_j, a)$ such that q_j' reversely simulates q_i', (2) $\mathcal{F}(q_i) = \mathcal{F}(q_j)$, and (3) $q_i = q_0$ implies $q_j = q_0$. After simplification using simulation relations, as in [23], a DPW may become nondeterministic. Therefore, the simplification by simulation can only be applied to the complement DPW.

Merging equivalent states. (`+E`). As for the conversion from an NPW to an NBW, a typical way in the literature is to go via an NRW [12,8]. We propose to go from NPW directly to NBW. Similar to the conversion from an NRW to an NBW in [12], we can nondeterministically guess the minimal even parity passed infinitely often in a run starting from some state. Once a run is guessed to pass a minimal even parity $2k$ infinitely often starting from a state s, every state t after s should have a parity greater than or equal to $2k$ and t is designated as an accepting state in the resulting NBW if it has parity $2k$. Moreover, we can make the resulting NBW smaller by merging states, with respect to an even parity $2k$, that have the same successors and have parities either all smaller than $2k$, all equal to $2k$, or all greater than $2k$. We can also start to guess the minimal even parity $2k$ starting from a state which has that parity.

5.2 For `Rank`

Maximizing Büchi acceptance set. (`+A`). As stated in Section 3, the ranks for the rank-based approach are bounded by $2(n - |\mathcal{F}|)$. The larger the $\mathcal{F}$ is, the fewer the ranks are. Thus, we propose to maximize the acceptance set of the input NBW without changing its language, states, or transition function. Given an NBW $A = (\Sigma, Q, q_0, \delta, \mathcal{F})$, we construct $A' = (\Sigma, Q, q_0, \delta, \mathcal{F}')$ with a larger acceptance set $\mathcal{F}' \supseteq \mathcal{F}$ such that $q \in \mathcal{F}'$ iff every elementary cycle containing q also contains at least one state in $\mathcal{F}$. Clearly the language of A' is the same as the language of A and we can take the complement of A' instead of A.

This heuristic can also be applied to other complementation approaches as it maximizes the acceptance set of the input NBW before complementation. We will show the improvement made by this heuristic for `Safra-Piterman`, `Rank`, and `Slice` later in Section 6.

5.3 For `Slice`

`Slice` constructs a complement with slices as states based on the evolution of a reduced split tree in the first phase, guesses the decoration for every vertex in a slice at some point, and verifies the guesses in the second phase. Intuitively, the decoration 1 indicates that a vertex must be in an infinite branch of a reduced split tree. The decoration 0 indicates that the descendants of a vertex must die out eventually before the next checkpoint. The decoration $*$ has the same meaning as 0 but the check is put on hold. In the second phase, `Slice` verifies two conditions: (1) a vertex decorated by 1 must have a right child decorated by 1, and (2) the left child of a vertex decorated by 1 and the children of a vertex decorated by 0 or $*$ must have a finite number of descendants.

Deterministic decoration. (`+D`). The first heuristic uses 1 to label vertices that *may* (rather than *must*) be in an infinite branch of a reduced split tree and only verifies the second condition in the second phase. All vertices could be decorated by 1 in the guesses. However, since the first evolution of the second phase always labels a left (accepting) child by 0 and a right (nonaccepting) child by 1, we actually decorate accepting vertices by 0 and nonaccepting vertices by 1 in the guesses. This heuristic will result in deterministic decoration. The only nondeterminism comes from choosing when to start decorating.

Reducing transitions. (`+R`). The second heuristic relies on the observation that if a run ends up in the empty sequence, a special slice denoted by $\perp$, the run will stay in $\perp$ forever and we never need to decorate the run because it can reach $\perp$ without any decoration. Thus we do not allow transitions from decorated slices other than $\perp$ to $\perp$ or from any slice to *doomed* slices; a slice is doomed if it is not $\perp$ and has no vertex labeled by 1, i.e., every run through a doomed slice is expected to reach $\perp$.

Merging adjacent vertices. (`+M`). The third heuristic recursively merges adjacent vertices decorated all by 0 or all by $*$. The observation is that they are all guessed to have a finite number of descendants and their successors will have the same decoration, either 0 or $*$.

6 Experimental Results

The heuristics proposed in Section 5 are also implemented in the GOAL tool. We use the same 11,000 automata as in Section 4 as the test bench. Since we do not propose any optimization heuristic for `Ramsey`, it is omitted in this experiment. The results showing the improvement made by the heuristics are listed in Table 2 where the Ratio columns are ratios with respect to the original construction and the other columns have the same meaning as they have in Section 4.

Compared with the original version for each construction, the experimental results in Table 2 show that (1) `Safra-Piterman+ASE` has 15 more unfinished tasks but creates almost one half of reachable states and live states, (2) the improvement made by `+A` is limited for `Safra-Piterman` and `Slice` but it is substantial for `Rank` in finishing 1,376 more tasks and avoiding the creation of around 2/3 dead states, (3) the heuristic `+D` is quite useful in reducing the reachable states down to 1/4 for `Slice` but makes more live states, and (4) `Slice+ADRM` finishes 6,116 more tasks and significantly reduces the reachable states to 1/10 and live states to one half.

Table 2. The results of comparing each construction with its improved versions

Constructions	Eff. Samples	S_R (Ratio)	S_L (Ratio)	S_L/S_R	T	M
`Safra-Piterman`	10,977	256.25 (1.00)	58.72 (1.00)	0.23	5	0
`Safra-Piterman+A`		228.40 (0.89)	54.33 (0.93)	0.24	5	0
`Safra-Piterman+S`		179.82 (0.70)	47.35 (0.81)	0.26	12	9
`Safra-Piterman+E`		194.95 (0.76)	45.47 (0.77)	0.23	11	0
`Safra-Piterman+ASE`		138.97 (0.54)	37.47 (0.64)	0.27	13	7
`Rank`	5,697	569.51 (1.00)	33.96 (1.00)	0.06	5,303	0
`Rank+A`		181.05 (0.32)	28.41 (0.84)	0.16	3,927	0
`Slice`	4,514	1,088.72 (1.00)	70.67 (1.00)	0.06	3,131	3,213
`Slice+A`		684.07 (0.63)	64.94 (0.92)	0.09	2,611	2,402
`Slice+D`		276.11 (0.25)	117.32 (1.66)	0.42	1,119	0
`Slice+R`		1,028.42 (0.94)	49.58 (0.70)	0.05	3,081	3,250
`Slice+M`		978.01 (0.90)	57.85 (0.82)	0.06	2,813	3,360
`Slice+ADRM`		102.57 (0.09)	36.11 (0.51)	0.35	228	0

Table 3. The results of comparing the three improved complementation constructions

Constructions	Eff. Samples	S_R (Win)	S_L (Win)	S_L/S_R	T	M
`Safra-Piterman+ASE`	7,045	49.94 (6,928.67)	21.38 (3,411.5)	0.43	13	7
`Rank+A`		428.61 (35.67)	41.80 (1,916.5)	0.10	3,927	0
`Slice+ADRM`		316.70 (80.67)	62.46 (1,717.0)	0.20	228	0
`Safra-Piterman+PASE`	7,593	44.84 (5,748.33)	19.50 (3,224)	0.43	4	0
`Rank+PA`		309.68 (910.33)	35.39 (2,340)	0.11	3,383	0
`Slice+PADRM`		270.68 (934.33)	53.67 (2,029)	0.20	216	0

We also compare the three constructions with all optimization heuristics in Section 5 based on 7,045 effective samples and list the results on the top of Table 3. The table shows that `Safra-Piterman+ASE` still outperforms the other two in the average state size and in running time. Table 3 also shows the following changes made by our heuristics in the comparison: (1) `Safra-Piterman+ASE` outperforms `Rank+A` in the number of smallest complements after pruning dead states, and (2) `Slice+ADRM` creates fewer reachable states than `Rank+A` in average, and finishes more tasks than `Rank+A`. As the heuristic of preminimization

applied to the input automata, denoted by `+P`, is considered to help the nondeterministic constructions more than the deterministic construction, we also compare the three constructions with preminimization and list the results in the bottom of Table 3. We only apply the preminimization implemented in the GOAL tool, namely the simplification by simulation in [23]. According to our experimental results, the preminimization does improve `Rank` and `Slice` more than `Safra-Piterman` in the complementation but does not close the gap too much between them in the comparison, though there are other preminimization techniques that we didn't implement and apply in the experiment.

7 Conclusion

We reviewed the state of Büchi complementation and examined the performance of the four complementation approaches by an experiment with a test set of 11,000 automata. We also proposed various optimization heuristics for three of the approaches and performed an experiment with the same test set to show the improvement. The experimental results show that the Safra-Piterman construction performs better than the other three in most cases in terms of time and state size. This is surprising and goes against the conventional wisdom that the nondeterministic approaches are better. The Ramsey-based construction is not competitive at all in complementation though it is competitive in universality and containment. The results also show that our heuristics substantially improve the Safra-Piterman construction and the slice-based construction in creating far fewer states. The rank-based construction and especially the slice-based construction can finish more complementation tasks with our heuristics. How the constructions scale with a growing state size, alphabet size, transition density, or other factors is not studied in this paper and is left as the future work.

References

1. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
2. Althoff, C.S., Thomas, W., Wallmeier, N.: Observations on determinization of Büchi automata. Theoretical Computer Science 363(2), 224–233 (2006)
3. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proceedings of the International Congress on Logic, Method, and Philosophy of Science 1960, pp. 1–12. Stanford University Press, Stanford (1962)
4. Doyen, L., Raskin, J.-F.: Antichains for the automata-based approach to model-checking. Logical Methods in Computer Science 5(1:5), 1–20 (2009)
5. Fogarty, S., Vardi, M.Y.: Büchi complementation and size-change termination. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 16–30. Springer, Heidelberg (2009)
6. Fogarty, S., Vardi, M.Y.: Efficient Büchi universality checking. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 205–220. Springer, Heidelberg (2010)

7. Friedgut, E., Kupferman, O., Vardi, M.Y.: Büchi complementation made tighter. International Journal of Foundations of Computer Science 17(4), 851–868 (2006)
8. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
9. Gurumurthy, S., Kupferman, O., Somenzi, F., Vardi, M.Y.: On complementing nondeterministic Büchi automata. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 96–110. Springer, Heidelberg (2003)
10. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 724–735. Springer, Heidelberg (2008)
11. Karmarkar, H., Chakraborty, S.: On minimal odd rankings for Büchi complementation. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 228–243. Springer, Heidelberg (2009)
12. King, V., Kupferman, O., Vardi, M.Y.: On the complexity of parity word automata. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 276–286. Springer, Heidelberg (2001)
13. Klarlund, N.: Progress measures for complementation of omega-automata with applications to temporal logic. In: FOCS, pp. 358–367. IEEE, Los Alamitos (1991)
14. Klein, J., Baier, C.: Experiments with deterministic ω-automata for formulas of linear temporal logic. Theoretical Computer Science 363(2), 182–195 (2006)
15. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Transactions on Computational Logic 2(3), 408–429 (2001)
16. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS, pp. 531–540. IEEE Computer Society, Los Alamitos (2005)
17. Michel, M.: Complementation is more difficult with automata on infinite words. Manuscript CNET, Paris (1988)
18. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. Theoretical Computer Science 141(1&2), 69–107 (1995)
19. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Logical Methods in Computer Science 3(3:5), 1–21 (2007)
20. Safra, S.: On the complexity of ω-automata. In: FOCS, pp. 319–327. IEEE, Los Alamitos (1988)
21. Schewe, S.: Büchi complementation made tight. In: STACS. LIPIcs, vol. 3, pp. 661–672. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2009)
22. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with appplications to temporal logic. TCS 49, 217–237 (1987)
23. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
24. Thomas, W.: Complementation of Büchi automata revisited. In: Jewels are Forever, pp. 109–120. Springer, Heidelberg (1999)
25. Tsai, M.-H., Fogarty, S., Vardi, M.Y., Tsay, Y.-K.: State of Büchi complementation (full version), `http://goal.im.ntu.edu.tw`
26. Tsay, Y.-K., Chen, Y.-F., Tsai, M.-H., Chan, W.-C., Luo, C.-J.: GOAL extended: Towards a research tool for omega automata and temporal logic. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 346–350. Springer, Heidelberg (2008)
27. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)

28. Vardi, M.Y.: Automata-theoretic model checking revisited. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 137–150. Springer, Heidelberg (2007)
29. Vardi, M.Y.: The Büchi complementation saga. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 12–22. Springer, Heidelberg (2007)
30. Vardi, M.Y., Wilke, T.: Automata: from logics to algorithms. In: Logic and Automata: History and Perspective. Texts in Logic and Games, vol. 2, pp. 629–736. Amsterdam University Press, Amsterdam (2007)
31. Yan, Q.: Lower bounds for complementation of omega-automata via the full automata technique. Logical Methods in Computer Science 4(1:5), 1–20 (2008)

Types of Trusted Information That Make DFA Identification with Correction Queries Feasible*

Cristina Tîrnăucă[1] and Cătălin Ionuţ Tîrnăucă[2]

[1] Departamento de Matemáticas, Estadística y Computación,
Universidad de Cantabria
Avda. de los Castros s/n, 39005 Santander, Spain
cristina.tirnauca@unican.es
[2] Research Group on Mathematical Linguistics, Universitat Rovira i Virgili
Av. Catalunya 35, 43002 Tarragona, Spain
catalinionut.tirnauca@estudiants.urv.cat

Abstract. In the query learning model, the problem of efficiently identifying a deterministic finite automaton (DFA) has been widely investigated. While DFAs are known to be polynomial time learnable with a combination of membership queries (MQs) and equivalence queries (EQs), each of these types of queries alone are not enough to provide sufficient information for the learner. Therefore, the possibility of having some extra-information shared between the learner and the teacher has been discussed. In this paper, the problem of efficient DFA identification with correction queries (CQs) - an extension of MQs - when additional information is provided to the learner is addressed. We show that knowing the number of states of the target DFA does not help (similar to the case of MQs or EQs), but other parameters such as the reversibility or injectivity degree are useful.

Keywords: deterministic finite automaton, correction query, injectivity degree.

1 Introduction

The problem of deterministic finite automaton (DFA) identification has a long history and dates back to E.M. Gold's pioneering paper [9]. There, he shows that in the framework of learning in the limit, regular languages are identifiable from informant but not from text. This is due to the fact that no superfinite class is learnable from text in the limit. Later on, Gold [10] and D. Angluin [1] proved that finding the smallest automaton consistent with a set of accepted and rejected strings is NP-complete.

The above-mentioned results make efficient DFA identification from given data a hard task. Therefore, the setting we are going to address in this paper is learning from requested data, the so-called *query learning model*. Typical

* This work was financially supported by *Ministerio de Educación y Ciencia de España* (MEC) grants JCDI-2009-04626 and MTM-2007-63422.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 272–281, 2011.

types of queries are membership, equivalence, subset, superset, disjointness and exhaustiveness queries (see [6] for further details and references).

In [5], a polynomial time algorithm that identifies any minimal complete DFA after asking a finite number of membership and equivalence queries is given. Later it is shown that neither membership queries (MQs) nor equivalence queries (EQs) alone are sufficient because even if the number of states of the DFA is given to the learner, one may still need, in the worst case, an exponential number of each of the two types of queries [6].

Actually, the idea of having extra-information provided by the teacher opens another whole research area:

> *"For many of the classes that can be taught efficiently, it is necessary in previous models to allow the teacher and learner to share a small amount of "trusted information", such as the size of the target function, since there is no other way to eliminate concepts from the given class that are more "complicated" than the target"* [11].

In the case of regular languages, if we denote by n the number of states of the minimal DFA and by k its reversibility degree (any regular language is k-reversible for some k), then neither n nor k can help in the DFA identification with MQs [2,15] or EQs [6,7]. On the other hand, if the auxiliary information consists of a set of strings guaranteed to reach every live state of the minimal DFA for the language, then the upper and lower bounds for the number of MQs needed are polynomial in n and the size of the given set of strings [2].

Other results addressing the same idea of additional information useful (or not) for polynomial identification with several types of queries concern k-term DNF formulas [4], context-free grammars [5], k-bounded context-free grammars [3], k-reversible languages [15], pattern languages [6] and singleton languages [6].

The main thrust of this paper is to analyze what kind of information helps the efficient identification of DFAs when data is requested in the form of correction queries (CQs) - an extension of the traditional MQs introduced in [8]. The main difference between CQs and MQs consists in the type of information revealed by the teacher for strings that do not belong to the target language. That is, instead of simply returning a "no", as in the case of MQs, the teacher's answer to a CQ provides the learner with some sort of "correction". Several types of corrections have been introduced so far (see [13] for a complete overview), but in the present contribution only the definition given in [8] is used: the correction is the smallest string in lex-length order that can be concatenated at the end of the queried datum to form a string in the target language.

The paper is organized as follows. Section 2 lists some basic notions about strings and automata extensively used through this article. Section 3 recalls the three concepts forming the core of the present work: the query learning model, the correction query and the injectivity degree. In Section 4 useful parameters for language learning with CQs are investigated. First of all, we argue why the number of states of the minimal DFA is a useless parameter. Next, we show that the injectivity degree is leading to correct polynomial time identification with CQs (Theorem 1). This is done by providing Algorithm 1 together with its

correctness and termination. In addition, the running time and query complexity is presented, and finally, a running example is described for a better understanding. Section 5 contains several concluding remarks and a short discussion on the optimality of Algorithm 1 and its further improvements.

2 Preliminaries

It is assumed the reader knows the basic facts about formal languages and automata, but we shall review the various basic notions and fix some of the general notation to be used throughout the paper. Expositions of the theory of automata and regular languages, as well as further references, can be found in [12] or [16], for example.

In what follows Σ is a finite *alphabet* of symbols. The set of all finite strings of symbols from Σ is denoted by Σ^*, and let λ be the empty string (i.e., the unique string of length 0). Subsets of Σ^* are called *languages*. For any finite set S, $S\Sigma = \{ua \mid u \in S, a \in \Sigma\}$, and $|S|$ denotes the cardinality of S, i.e., its number of elements. The length of a string w is denoted by $|w|$, and the concatenation of two strings u and v by uv. If $w = uv$ for some $u, v \in \Sigma^*$, then u is a *prefix* of w and v a *suffix* of w. If Σ is a totally ordered set, then u is smaller than v in *lex-length order* if either $|u| < |v|$, or $|u| = |v|$ and u is smaller than v lexicographically.

Let $\Sigma^{\leq k} = \{w \in \Sigma^* \mid |w| \leq k\}$ and $\Sigma^k = \{w \in \Sigma^* \mid |w| = k\}$. For every $L \subseteq \Sigma^*$ and $u \in \Sigma^*$, the set of all prefixes of L is $Pref(L) = \{w \mid \exists v \in \Sigma^*$ such that $wv \in L\}$, and the *left-quotient* of L and u is the set $Tail_L(u) = \{v \mid uv \in L\}$. Note that $Tail_L(u) \neq \emptyset$ if and only if $u \in Pref(L)$.

A *deterministic finite automaton* (DFA) is a device $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states*, Σ is a finite alphabet, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and δ is a partial function, called *transition function*, that maps $Q \times \Sigma$ to Q. This function can be extended to strings by setting $\delta(q, \lambda) = q$ for all $q \in Q$, and $\delta(q, ua) = \delta(\delta(q, u), a)$ for all $q \in Q$, $u \in \Sigma^*$ and $a \in \Sigma$. The language *accepted* by $\mathcal{A}$ is the set $L(\mathcal{A}) = \{u \in \Sigma^* \mid \delta(q_0, u) \in F\}$. Such a DFA $\mathcal{A}$ is *complete* if for every $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ is defined, i.e., δ is a total function. For any two DFAs $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ and $\mathcal{A}' = (Q', \Sigma, \delta', q_0', F')$, $\varphi : Q \to Q'$ is an *automata morphism* if φ is a (well-defined) function such that $\varphi(q_0) = q_0'$, $\varphi(F) \subseteq F'$ and $\varphi(\delta(q, a)) = \delta'(\varphi(q), a)$ for all $q \in Q$ and $a \in \Sigma$. Moreover, φ is said to be an *automata isomorphism* if it is a bijection and $\varphi(F) = F'$.

A language $L \subseteq \Sigma^*$ is *regular* if $L = L(\mathcal{A})$ for some DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$. Recall that for every regular language L, there exists a minimum state DFA $\mathcal{A}_L$ such that $L(\mathcal{A}_L) = L$ (cf. [12, pp. 65–71]).

Given a language $L \subseteq \Sigma^*$, the relation $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ is defined by: $u_1 \equiv_L u_2$ if and only if for all u in Σ^*, $u_1 u \in L \Leftrightarrow u_2 u \in L$. Obviously, $\equiv_L$ is an equivalence relation, and thus it divides the set of all finite strings into one or more equivalence classes. The Myhill-Nerode Theorem states that the number of equivalence classes of $\equiv_L$ (also called the *index* of L and denoted by $index(L)$)

is equal to the number of states of $\mathcal{A}_L$. As a direct consequence, a language L is regular if and only if the index of L is finite.

3 Query Learning

In the query learning model [5] a learner has access to an oracle that truthfully answers queries of a specified kind. A *query learner* M is an algorithmic device that, depending on the reply to the previous queries, either computes a new query, or returns a hypothesis and halts. Given an indexable class $\mathcal{C} = (L_i)_{i\geq 1}$ and a language L in $\mathcal{C}$, we say that M *learns* L if, after asking a finite number of queries, it outputs an index i such that $L_i = L$. Moreover, M learns $\mathcal{C}$ using some type of queries if it learns every $L \in \mathcal{C}$ using queries of the specified type.

Various types of queries have been introduced so far, but the first and most investigated ones are MQs (*is this string in the target language?*) and EQs (*is this the target language?*). As we already mentioned in the Introduction, our paper is about learning with CQs. In the case of a CQ for the target language L, the input is a string w and the answer is the smallest string (in lex-length order) of the set $Tail_L(w)$ if $w \in Pref(L)$, and the special symbol $\theta \notin \Sigma$ otherwise. We denote the correction of a string w with respect to the language L by $C_L(w)$.

Cf. [14], the *injectivity degree* of a regular language $L \subseteq \Sigma^*$ is

$$InjDeg(L) := index(L) - |\{C_L(u) \mid u \in \Sigma^*\}| \quad .$$

We say that L is k-*injective* if $InjDeg(L) = k$. The class of all regular languages having injectivity degree k is denoted by $k\mathcal{I}nj$.

For all languages $L \subseteq \Sigma^*$ with $index(L) \geq 2$, $InjDeg(L)$ can take values between 0 and $index(L)-2$. We give two examples of classes with extreme values of the injectivity degree. Let $\mathcal{S}_k = (L_w)_{w\in\Sigma^k}$ with $L_w = \{w\}$ and $\bar{\mathcal{S}}_k = (\bar{L}_w)_{w\in\Sigma^k}$ with $\bar{L}_w = \Sigma^*\backslash\{w\}$. It is easy to check that for every language $L = L_w$ in $\mathcal{S}_k$, the automaton $\mathcal{A}_L$ has $n = k + 2$ states and the set $\{C_L(u) \mid u \in \Sigma^*\}$ has exactly n elements (if $w = a_1a_2\dots a_k$ then the set of possible corrections is $\{\lambda, a_k, a_{k-1}a_k, \dots, a_1a_2\dots a_k, \theta\}$). That means $InjDeg(L) = 0$. On the other hand, for every $L = \bar{L}_w$ in $\bar{\mathcal{S}}_k$, $\mathcal{A}_L$ has $n = k + 2$ states as well, but the set $\{C_L(u) \mid u \in \Sigma^*\}$ contains only 2 elements: λ and a, where a is the smallest letter of Σ. Therefore, $InjDeg(L)$ in this case is $n - 2$.

In [14] it was shown that 0-injective languages are learnable with CQs in polynomial time. In the next section we generalize this result: the algorithm in [14] is basically our Algorithm 1 for the case $k = 0$.

4 Useful Parameters for Language Learning with CQs

First, we should mention that the number of states of the minimal DFA is a useless parameter [13, p.72]. The argument used by Angluin for the case of MQs and EQs [6] cannot be employed for CQs, because in order to learn the class $\mathcal{S}_k$ of singleton languages of fixed length, one CQ is enough to disclose the

target. However, one may still need an exponential number of CQs to learn a given DFA even when we know its number of states. Indeed, consider the class $\bar{\mathcal{S}}$ defined above. Any consistent teacher would provide the same answer λ to all but one element of the set Σ^k. Therefore, no matter what learning strategy we choose, we might need to ask $|\Sigma| + \ldots + |\Sigma|^k$ CQs in the worst case.

On the other hand, the degree of reversibility does help the learning process: in [15], a polynomial time algorithm for learning the class of k-reversible languages with CQs is provided.

In the sequel, we show that once we know the degree of injectivity k, we can efficiently identify the class $k\mathcal{I}nj$ using a linear number of CQs. Note that $k\mathcal{I}nj$ is not polynomial time learnable with MQs: an exponential number of MQs is needed, in the worst case, to learn the class $\mathcal{S}_k$ of all singleton languages of fixed length [2].

4.1 Learning k-Injective Languages with CQs

The algorithm we present for learning the class $k\mathcal{I}nj$ is based on the following result, given without proof due to space restrictions.

Proposition 1. *Let $L \subseteq \Sigma^*$ be a regular language of injectivity degree k. Then for every $u, v \in \Sigma^*$ with $u \not\equiv_L v$, there exists w in $\Sigma^{\leq k}$ such that $C_L(uw) \neq C_L(vw)$.*

The algorithm follows the lines of L* [5]. We have an *observation table*, denoted (S, E, C), where lines are indexed by the elements of a finite set $S \cup S\Sigma$, columns are indexed by the elements of the set E, which in our case equals $\Sigma^{\leq k}$, and the entry of the table at row u and column v is $C_L(uv)$. One important difference between our algorithm and the original one is that in L*, E contains only one element in the beginning, and it is gradually enlarged whenever an equivalence query is answered with a counterexample. We start with $S = \{\lambda\}$.

For every u in $S \cup S\Sigma$, we define the function $row(u) : E \to \Sigma^* \cup \{\theta\}$ by setting $row(u)(v) = C_L(uv)$. Then, $row(S) = \{row(u) \mid u \in S\}$. The observation table (S, E, C) is called *closed* if for all $u \in S$ and $a \in \Sigma$, there exists $u' \in S$ such that $row(u') = row(ua)$, and *consistent* if for all $u_1, u_2 \in S$, $row(u_1) \neq row(u_2)$. We will see that, as opposed to L*, our tables are always consistent.

Algorithm 1. An algorithm for learning the class $k\mathcal{I}nj$ with CQs

1: $S := \{\lambda\}$, $E := \Sigma^{\leq k}$
2: update the table by asking CQs for all strings in $(S \cup S\Sigma)E$
3: **while** (S, E, C) is not closed **do**
4: find $u \in S$ and $a \in \Sigma$ such that $row(ua) \notin row(S)$
5: add ua to S
6: update the table by asking CQs for all strings in $\{uaa'v \mid a' \in \Sigma, v \in E\}$
7: **end while**
8: output $\mathcal{A}(S, E, C)$ and halt.

For any closed and consistent observation table (S, E, C), one can construct the automaton $\mathcal{A}(S, E, C) = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{row(u) \mid u \in S\}$, $q_0 = row(\lambda)$, $F = \{row(u) \mid u \in S \text{ and } C_L(u) = \lambda\}$, and $\delta(row(u), a) = row(ua)$ for all $u \in S$ and $a \in \Sigma$. Note that $\mathcal{A}(S, E, C)$ is well defined because for every $u \in S$ and $a \in \Sigma$, there exists a unique u' in S such that $row(ua) = row(u')$. For every $u \in S \cup S\Sigma$, it can be easily shown by induction on the length of the string u that $\delta(q_0, u) = row(u)$.

Since Algorithm 1 adds to S only elements with distinct row values, the table (S, E, C) is always consistent. The following lemma witnesses that as long as the cardinality of the set S is smaller than the number of states of the target automaton, (S, E, C) is not closed.

Lemma 1. *Let n be the number of states of the automaton $\mathcal{A}_L$. If $|S| < n$, then (S, E, C) is not closed.*

Proof. We assume there exists $m < n$ such that $|S| = m$ and the table (S, E, C) is closed. Let $\mathcal{A}(S, E, C) = (Q, \Sigma, \delta, q_0, F)$, and let $\mathcal{A}_L = (Q', \Sigma, \delta', q'_0, F')$ be the minimal complete automaton accepting L.

We define the function $\varphi : Q \to Q'$ by setting $\varphi(row(u)) := \delta'(q'_0, u)$. Clearly, φ is well defined because there are no two strings u_1, u_2 in S such that $row(u_1) = row(u_2)$. Moreover, it is injective since $\varphi(row(u_1)) = \varphi(row(u_2))$ implies $\delta'(q'_0, u_1) = \delta'(q'_0, u_2)$, that is, $u_1 \equiv_L u_2$, which entails $row(u_1) = row(u_2)$. Next, we show that φ is an automata morphism from $\mathcal{A}(S, E, C)$ to $\mathcal{A}_L$. Clearly, $\varphi(q_0) = q'_0$ and $\varphi(F) \subseteq F'$.

It remains to prove $\varphi(\delta(row(u), a)) = \delta'(\varphi(row(u)), a)$ for all $u \in S$ and $a \in \Sigma$. To this end, we have $\varphi(\delta(row(u), a)) = \varphi(row(ua)) = \varphi(row(v)) = \delta'(q'_0, v)$ for some v in S such that $row(ua) = row(v)$ (the table is closed). Moreover, $\delta'(\varphi(row(u)), a) = \delta'(\delta'(q'_0, u), a) = \delta'(q'_0, ua)$. To conclude the proof, it is enough to see that $\delta'(q'_0, v) = \delta'(q'_0, ua)$ (since $row(v) = row(ua)$ implies, by Proposition 1, that $v \equiv_L ua$).

Hence, we have constructed an injective automata morphism from $\mathcal{A}(S, E, C)$ to $\mathcal{A}_L$ such that $|Q| = m < n = |Q|$. Since both $\mathcal{A}(S, E, C)$ and $\mathcal{A}_L$ are complete automata, we get a contradiction. □

Next we show that the algorithm runs in polynomial time, and it terminates with the minimal automaton for the target language as its output, therefore justifying the following theorem.

Theorem 1. *The class $k\mathcal{I}nj$ is polynomial time learnable with CQs.*

Note that when k is given, the learner no longer needs to ask EQs.

Correctness and Termination. We have seen that as long as $|S| < n$, the table is not closed, so there will always be an u in S and a symbol a in Σ such that $row(ua) \notin row(S)$. Since the cardinality of the set S is initially 1 and increases by 1 within each "while" loop (lines 3–7 of Algorithm 1), it will eventually be n, and hence the algorithm is guaranteed to terminate. The correctness of Algorithm 1 is given by the following lemma.

Lemma 2. *Let n be the number of states of the automaton $\mathcal{A}_L$. If $|S| = n$, then (S, E, C) is closed and $\mathcal{A}(S, E, C)$ is isomorphic to $\mathcal{A}_L$.*

Proof. Indeed, if $|S| = n$, then the set $\{row(u) \mid u \in S\}$ has cardinality n because the elements of S have distinct row values. Thus, for every $u \in S$ and $a \in \Sigma$, $row(ua) \in row(S)$ (otherwise $row(ua)$ would be the $(n+1)^{\text{th}}$ equivalence class of the set of all equivalence classes induced by $\equiv_L$ on Σ^*), and hence the table is closed.

To see that $\mathcal{A}(S, E, C)$ and $\mathcal{A}_L$ are isomorphic, let us take $\mathcal{A}(S, E, C) = (Q, \Sigma, \delta, q_0, F)$, $\mathcal{A}_L = (Q', \Sigma, \delta', q_0', F')$, and the function $\varphi : Q \to Q'$ defined as in the proof of Lemma 1. Using the same arguments, it can be shown that φ is a well-defined and injective automata morphism. Since the two automata have the same number of states, φ is also surjective, and hence bijective. It is quite easy to check that $\varphi(F) = F'$, and hence $\mathcal{A}(S, E, C)$ and $\mathcal{A}_L$ are isomorphic. □

Time Analysis and Query Complexity. Let us now discuss the time complexity of Algorithm 1. While the cardinality of S is smaller than n, where n represents the number of states of $\mathcal{A}_L$, the algorithm searches for a string u in S and a symbol a in Σ such that $row(ua)$ is distinct from all $row(v)$ with $v \in S$. This can be done using at most $|S|^2 \cdot |\Sigma| \cdot |E|$ operations: there are $|S|$ possibilities for choosing u (and the same number for v), $|\Sigma|$ for choosing a, and $|E|$ operations to compare $row(ua)$ with $row(v)$. For $|\Sigma| = l$, $|E| = 1 + l + l^2 + \ldots + l^k$ and thus the total running time of the "while" loop can be bounded by $(1^2 + 2^2 + \ldots + (n-1)^2) \cdot l \cdot (1 + l + l^2 + \ldots + l^k)$. Note that by "operations" we mean string comparisons, since they are generally acknowledged as being the most costly tasks.

On the other hand, to construct $\mathcal{A}(S, E, C)$ we need n comparisons for determining the final states, and another $n^2 \cdot |\Sigma| \cdot |E|$ operations for constructing the transition function. This means that the total running time of Algorithm 1 is bounded by $n + l \cdot \frac{l^{k+1}-1}{l-1} \cdot \frac{n(n+1)(2n+1)}{6}$, that is $O(n^3)$, since k is a constant for the class to be learned.

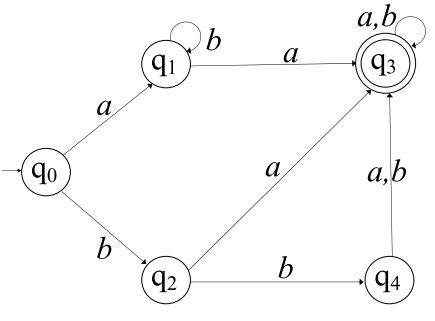


Fig. 1. Minimal complete DFA for the language $L = (ab^*a + ba + bb(a + b))(a + b)^*$

As for the number of queries asked by the algorithm, it can be bounded by $|S \cup S\Sigma| \cdot |E|$ (i.e., by the size of the final observation table), so the query complexity of the algorithm is $O(n)$.

Running Example. Firstly, let $L = (ab^*a + ba + bb(a + b))(a + b)^* \subseteq \{a, b\}^*$ be the target language. The minimal DFA $\mathcal{A}_L$ is depicted in Fig. 1.

The algorithm starts with $S = \{\lambda\}$, $E = \{\lambda, a, b, aa, ab, ba, bb\}$ and the observation table illustrated in Table 1. Since the value of $row(a)$ is distinct from the value of $row(\lambda)$, this table is not closed. Consequently, the algorithm proceeds by adding the string a to S, updating Table 1 and obtaining Table 2.

Table 1. $S = \{\lambda\}$

First Table		E						
		λ	a	b	aa	ab	ba	bb
S	λ	aa	a	a	λ	a	λ	a
$S\Sigma\backslash S$	a	a	λ	a	λ	λ	λ	a
	b	a	λ	a	λ	λ	λ	λ

Table 2. $S = \{\lambda, a\}$

Second Table		E						
		λ	a	b	aa	ab	ba	bb
S	λ	aa	a	a	λ	a	λ	a
	a	a	λ	a	λ	λ	λ	a
$S\Sigma\backslash S$	b	a	λ	a	λ	λ	λ	λ
	aa	λ	λ	λ	λ	λ	λ	λ
	ab	a	λ	a	λ	λ	λ	a

Table 2 is not closed because the value of $row(b)$ is different from the value of $row(a)$. Next, the algorithm adds the string b to S and updates the table. We get Table 3, which is not closed (e.g., $row(aa) \neq row(a)$). Repeating analogous steps, the string aa is set to be in S and the table is updated to Table 4.

Table 3. $S = \{\lambda, a, b\}$

Third Table		E						
		λ	a	b	aa	ab	ba	bb
S	λ	aa	a	a	λ	a	λ	a
	a	a	λ	a	λ	λ	λ	a
	b	a	λ	a	λ	λ	λ	λ
$S\Sigma\backslash S$	aa	λ	λ	λ	λ	λ	λ	λ
	ab	a	λ	a	λ	λ	λ	a
	ba	λ	λ	λ	λ	λ	λ	λ
	bb	a	λ	λ	λ	λ	λ	λ

Table 4. $S = \{\lambda, a, b, aa\}$

Fourth Table		E						
		λ	a	b	aa	ab	ba	bb
S	λ	aa	a	a	λ	a	λ	a
	a	a	λ	a	λ	λ	λ	a
	b	a	λ	a	λ	λ	λ	λ
	aa	λ	λ	λ	λ	λ	λ	λ
$S\Sigma\backslash S$	ab	a	λ	a	λ	λ	λ	a
	ba	λ	λ	λ	λ	λ	λ	λ
	bb	a	λ	λ	λ	λ	λ	λ
	aaa	λ	λ	λ	λ	λ	λ	λ
	aab	λ	λ	λ	λ	λ	λ	λ

But $row(bb) \neq row(\lambda)$, so the algorithm adds bb to S and updates Table 4, getting this way Table 5, which is both closed and consistent.

Finally, the algorithm outputs the automaton $\mathcal{A}(S, E, C)$, which is precisely the one represented in Fig. 1, and halts.

Table 5. $S = \{\lambda, a, b, aa, bb\}$

Fifth Table		E							State
		λ	a	b	aa	ab	ba	bb	
S	λ	aa	a	a	λ	a	λ	a	q_0
	a	a	λ	a	λ	λ	λ	a	q_1
	b	a	λ	a	λ	λ	λ	λ	q_2
	aa	λ	λ	λ	λ	λ	λ	λ	q_3
	bb	a	λ	λ	λ	λ	λ	λ	q_4
$S\Sigma\backslash S$	ab	a	λ	a	λ	λ	λ	a	q_1
	ba	λ	λ	λ	λ	λ	λ	λ	q_3
	aaa	λ	λ	λ	λ	λ	λ	λ	q_3
	aab	λ	λ	λ	λ	λ	λ	λ	q_3
	bba	λ	λ	λ	λ	λ	λ	λ	q_3
	bbb	λ	λ	λ	λ	λ	λ	λ	q_3

5 Concluding Remarks

We have addressed the problem of polynomial time identification of DFAs in the query learning model where the requested data is in the form of correction queries. Although at a first glance CQs seem to be just a finite collection of MQs, this is not exactly the case as indicated by various arguments (see [13] for a detailed discussion). In this paper we have seen yet another case in which some extra-information provided to a CQ learner does help the learning process whereas the same information is useless for an MQ learner or an EQ learner.

Please note that Algorithm 1 provided here is not intended to be optimal. One can easily find smarter implementations. For example, instead of having the whole set $\Sigma^{\leq k}$ as experiments, the algorithm could start with $E = \{\lambda\}$ and then gradually add, in lex-length order, only those w in $\Sigma^{\leq k}$ that would create in $S\Sigma\backslash S$ a string with different row values. As for the stopping condition, one may count the number of elements in $|S| - |\{C_L(u) \mid u \in S\}|$ (which is initially 0) and halt whenever this number reaches k and the table becomes closed again. Also observe that although this new algorithm would work better on numerous DFAs, it still shares the same worst case complexity due to the fact that sometimes, in order to distinguish two states, the shortest string needed is of maximal length (see the strings a and b in the DFA depicted in Fig. 1). Of course, for any other strategy of choosing how elements in $\Sigma^{\leq k}$ should be added to E, one can always find an adversary example.

Another possible improvement consists of using an ordered container or a hash map to store the set of lines in the observation table (the searching time becomes $|E| \log(|S|)$ instead of $|E||S|$).

Acknowledgments. The authors are grateful to the anonymous reviewers for their remarks and suggestions, which significantly improved the quality of the exposition.

References

1. Angluin, D.: On the complexity of minimum inference of regular sets. Information and Control 39(3), 337–350 (1978)
2. Angluin, D.: A note on the number of queries needed to identify regular languages. Information and Control 51(1), 76–87 (1981)
3. Angluin, D.: Learning k-bounded context-free grammars. Technical Report TR-557, Yale University, New Haven, Conn. (1987)
4. Angluin, D.: Learning k-term DNF formulas using queries and counter-examples. Technical Report TR-559, Yale University, New Haven, Conn. (1987)
5. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
6. Angluin, D.: Queries and concept learning. Machine Learning 2(4), 319–342 (1988)
7. Angluin, D.: Negative results for equivalence queries. Machine Learning 5(2), 121–150 (1990)
8. Becerra-Bonache, L., Dediu, A.H., Tîrnăucă, C.: Learning DFA from correction and equivalence queries. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 281–292. Springer, Heidelberg (2006)
9. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
10. Gold, E.M.: Complexity of automaton identification from given data. Information and Control 37(3), 302–320 (1978)
11. Goldman, S.A., Mathias, H.D.: Teaching a smarter learner. Journal of Computer and System Sciences 52(2), 255–267 (1996)
12. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
13. Tîrnăucă, C.: Language Learning with Correction Queries. Ph.D. thesis, Universitat Rovira i Virgili, Tarragona, Spain (2009)
14. Tîrnăucă, C., Knuutila, T.: Efficient language learning with correction queries. Technical Report 822, Turku Center for Computer Science (2007)
15. Tîrnăucă, C., Knuutila, T.: Polynomial time algorithms for learning k-reversible languages and pattern languages with correction queries. In: Hutter, M., Servedio, R.A., Takimoto, E. (eds.) ALT 2007. LNCS (LNAI), vol. 4754, pp. 264–276. Springer, Heidelberg (2007)
16. Yu, S.: Regular languages. In: Salomaa, A., Rozenberg, G. (eds.) Handbook of Formal Languages. Word, Language, Grammar, vol. 1, ch. 2, pp. 41–110. Springer, Berlin (1997)

Compressing Regular Expressions' DFA Table by Matrix Decomposition

Yanbing Liu[1,2,3], Li Guo[1,3], Ping Liu[1,3], and Jianlong Tan[1,3]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190
[2] Graduate School of Chinese Academy of Sciences, Beijing, 100049
[3] National Engineering Laboratory for Information Security Technologies, 100190
liuyanbing@software.ict.ac.cn,
{guoli,liuping,tjl}@ict.ac.cn

Abstract. Recently regular expression matching has become a research focus as a result of the urgent demand for Deep Packet Inspection (DPI) in many network security systems. Deterministic Finite Automaton (DFA), which recognizes a set of regular expressions, is usually adopted to cater to the need for real-time processing of network traffic. However, the huge memory usage of DFA prevents it from being applied even on a medium-sized pattern set. In this article, we propose a matrix decomposition method for DFA table compression. The basic idea of the method is to decompose a DFA table into the sum of a row vector, a column vector and a sparse matrix, all of which cost very little space. Experiments on typical rule sets show that the proposed method significantly reduces the memory usage and still runs at fast searching speed.

1 Introduction

Recent years, regular expression matching has become a research focus in network security community. This interest is motivated by the demand for Deep Packet Inspection (DPI), which inspects not only the headers of network packets but also the payloads. In network security systems, signatures are represented as either exact strings or complicated regular expressions, and the number of signatures is quite large. Considering the requirement on real-time processing of network traffic in such systems, Deterministic Finite Automaton (DFA) is usually adopted to recognize a set of regular expressions. However, the combined DFA for regular expressions might suffer from the problem of exponential blow-up, and the huge memory usage prevents it from being applied even on a medium-sized pattern set. Therefore it's necessary to devise compression methods to reduce DFA's space so that it can reside in memory or high speed CPU caches.

In this article, we propose a matrix decomposition method for DFA table compression. Matrix decomposition has been widely studied and used in many fields, but it has not yet been considered for DFA table compression. We treat the state transition table of DFA as a matrix, and formulate a scheme for DFA table compression from the angle of matrix decomposition. The basic idea of our

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 282–289, 2011.

method is to decompose a DFA table into three parts: a row vector, a column vector and a residual sparse matrix, all of which cost very little space. We test our method on typical regular expression rule sets and the results show that the proposed method significantly reduces the memory usage and still runs at fast searching speed comparable to that of the original DFA.

The rest of this paper is organized as follows. We first summarize related work in DFA compression area in section 2. And then we formulate a matrix decomposition problem for DFA compression in section 3.1. After that, we propose an iterative algorithm for matrix decomposition and DFA compression in section 3.2. Finally, we carry out experiments with the proposed method and report the results in section 4. Section 5 concludes the paper.

2 Related Work

Lots of theoretic and algorithmic results on regular expression matching have been achieved since 1960s [1–6]. To bridge the gap between theory and practice, recent years there are great interests in implementing fast regular expression matching in real-life systems [7–14]. The large memory usage and potential state explosion of DFA are the common concerns of many researchers.

Yu et al.[7] exploit rewrite rules to simplify regular expressions, and develop a grouping method that divides a set of regular expressions into several groups so that they can be compiled into medium-sized DFAs. The rewrite rules work only if the non-overlapping condition is satisfied.

Kumar et al. [8] propose Delayed Input DFA which uses *default transition* to eliminate redundant transitions, but the time of state switching per text character increases proportionally.

Becchi et al. [9] propose a compression method that results in at most $2N$ state traversals when processing an input text of length N. It takes advantage of state distance to achieve high compressibility comparable to that of the Delayed Input DFA method.

Ficara et al. [10] devise the method δFA to eliminate redundant transitions in DFA. The idea is based on the observation that adjacent states in DFA traversing share the majority of the next-hop states associated with the same input characters, therefore the transitions of current state can be retrieved from its predecessor's transition table dynamically. However, the update of a local state transition table is time-consuming.

Simth et al. [11] introduce XFA to handle two special classes of regular expressions that suffer from the exponential explosion problem. XFA augments DFA by attaching counters to DFA states to memorize additional information. This method needs to update a set of counters associated with each state during traversing, and therefore it is not practical for software implementation.

In short, most of the above methods make use of space-time tradeoff: reducing space usage at the cost of increasing running time. Though these methods are efficient in particular environments, better space-time tradeoff techniques are still need to be studied.

3 A Matrix Decomposition Method for DFA Compression

In this section, we first formulate a matrix decomposition problem: Additive Matrix Decomposition. And then we propose an iterative algorithm to solve the stated problem. Based on the matrix decomposition, a DFA compression scheme is naturally followed.

3.1 Problem Formulation: Additive Matrix Decomposition

The state transition table of a DFA can be treated as an $m \times n$ matrix A, where m is the number of states and n is the cardinality of alphabet Σ. Matrix element $A[i, j]$ (or $A_{i,j}$) defines the state switching from current state i to the next state through character label j.

The basic idea of our method is approaching the DFA matrix A by a special matrix D (that can be stored with little space) so that the residual matrix $R = A - D$ is as sparse as possible. By replacing the original DFA matrix with the special matrix D and the residual sparse matrix R, a space compression scheme sounds plausible. We formulate our idea as the following matrix decomposition problem:

Additive Matrix Decomposition. *Let X be a column vector of size m and Y be a row vector of size n. Let D be the $m \times n$ matrix induced by X and Y with $D[i, j] = X[i] + Y[j]$ $(1 \leq i \leq m,\ 1 \leq j \leq n)$. Now given an $m \times n$ matrix A, find X and Y such that the number of zero elements in the residual matrix $R = A - D = [A[i, j] - X[i] - Y[j]]$ is maximized.*

According to above matrix decomposition, DFA matrix A can be represented with a column vector X, a row vector Y, and a residual matrix R. Since $A[i, j] = X[i] + Y[j] + R[i, j]$, state switching in DFA is still $O(1)$ as long as accessing an element in the residual sparse matrix R is accomplished in $O(1)$ time.

For the purpose of high compressibility and fast access time, the residual matrix R should be as sparse as possible. Space usage of the proposed scheme consists of the size of X, the size of Y, and the number of nonzero elements in the residual matrix $R = A - D$, resulting in the compression ratio $\frac{m+n+\text{nonzero}(R)}{mn}$. This metric is used to evaluate our method's compression efficiency in section 4.

3.2 Iterative Algorithm for Additive Matrix Decomposition

We present here an iterative algorithm to find the vectors X and Y that maximize the number of zero elements in the residual matrix R.

We start with the observation that if vectors X and Y are the optimal vectors to the additive matrix decomposition problem, then the following necessary constraints must be satisfied:

1. For any $1 \leq i \leq m$, $X[i]$ is the most frequent element in multiset $D_{i\cdot} = \{A[i, j] - Y[j] \mid 1 \leq j \leq n\}$.
2. For any $1 \leq j \leq n$, $Y[j]$ is the most frequent element in multiset $D_{\cdot j} = \{A[i, j] - X[i] \mid 1 \leq i \leq m\}$.

The above constrains are easy to obtain. For fixed $Y[j]$, if $X[i]$ is not the most frequent element in $D_{i.}$, then we can increase the number of zero elements in R by simply replacing $X[i]$ with the most frequent element in $D_{i.}$. Constrains hold for Y likewise.

We devise an iterative algorithm based on the above constraints to compute X and Y. The elements of X and Y are firstly initialized to random seeds. Then we iteratively compute X from current Y and compute Y from current X until the above constraints are all satisfied. The number of zero elements in R is increasing during each iteration, and therefore the algorithm terminates in finite steps. In practice this algorithm usually terminates in 2 or 3 iterations. Since the constraints are not sufficient conditions, our iterative algorithm might not converge to a global optimal solution. Fortunately, the algorithm usually produces fairly good results.

The iterative procedure for computing X and Y is described in algorithm 1.

4 Experiment and Evaluation

We carry out experiments on several regular expression rule sets and compare our method (CRD, Column-Row Decomposition) with the original DFA as well as the δFA method[10] in terms of compression efficiency and searching time. The *CHAR-STATE* technique in δFA is not implemented because it is not practical for software implementation.

The experiments are carried out on regular expression signatures obtained from several open-source systems, including: L7-filter[16], Snort[17], BRO[18]. We also generate 6 groups of synthetic rules according to the classification proposed by Fang et.al[7], who categorize regular expressions into several classes with different space complexity.

Since the DFA for a set of regular expressions usually suffers from the state blow-up problem, it is usually hard to generate a combined DFA for a whole large rule set. We use the *regex-tool*[19] to partition a large rule set into several parts and to generate a moderate-sized DFA for each subset. In experiments the L7-filter rule set is divided into 8 subsets, and the Snort rule set is divided into 3 parts. Details of the rule sets are described in table 1.

4.1 Compression Efficiency Comparison

This section compares the space usage of our method CRD with that of the original DFA and the δFA. We also compare our method CRD with its two simplified versions: DefaultROW and DefaultCOL. DefaultROW (or DefaultCOL) corresponds to set the vector Y (or X) in CRD to zero, and to extract the most frequent element in each row (or column) as a default state.

We use the term *compression ratio* to evaluate the methods' compression efficiency. For the original DFA, its compression ratio is always 1.0. For δFA, its compression ratio is the percent of nonzero elements in the final residual sparse matrix. For our method CRD, its compression ratio is defined as $\frac{m+n+\text{nonzero}(R)}{mn}$.

Algorithm 1. Decompose an $m \times n$ matrix A into a column vector X with size m, a row vector Y with size n, and an $m \times n$ sparse matrix R. $A[i, j] = X[i] + Y[j] + R[i, j]$. Let $n(x, S)$ denote the number of occurrences of x in a multiset S.

```
1: procedure MatrixDecomposition(A, m, n)
2:     for i ← 1, m do
3:         X[i] ←rand()
4:     end for
5:     for j ← 1, n do
6:         Y[j] ←rand()
7:     end for
8:     repeat
9:         changed ←FALSE
10:        for i ← 1, m do
11:            x ← the most frequent element in multiset D_i. = {A[i, j] − Y[j] | 1 ≤ j ≤ n}
12:            if n(x, D_i.) > n(X[i], D_i.) then
13:                X[i] ← x
14:                changed ←TRUE
15:            end if
16:        end for
17:        for j ← 1, n do
18:            y ← the most frequent element in multiset D_.j = {A[i, j] − X[i] | 1 ≤ i ≤ m}
19:            if n(y, D_.j) > n(Y[j], D_.j) then
20:                Y[j] ← y
21:                changed ←TRUE
22:            end if
23:        end for
24:    until changed =FALSE
25:    R ← [A[i, j] − X[i] − Y[j]]_{m×n}
26:    return (X, Y, R)
27: end procedure
```

Algorithm 2. State switching in our DFA table compression scheme

```
1: procedure NextState(s, c)
2:     t ← X[s] + Y[c]
3:     if BloomFilter. test(s, c) = 1 then
4:         t ← t + SparseMatrix. get(s, c)
5:     end if
6:     return t
7: end procedure
```

Table 1 presents the compression ratio of the algorithms on typical rule sets. We can see that our method achieves better compressibility on L7-filter, BRO and synthetic rules, whereas δFA performs better on Snort rules. Of all the 18 groups of rules, CRD outperforms δFA on 14 rule sets. We can also see that CRD combines the merits of both DefaultROW and DefaultCOL, and performs better than these two simplified versions except on the rule set *Synthetic_1*.

Table 1. Compression ratio of the algorithms on L7-filter, Snort, BRO and synthetic rule sets

Rule set	# of rules	# of states	δFA	CRD	DefaultROW	DefaultCOL
L7_1	26	3172	0.634964	**0.226984**	0.232905	0.817074
L7_2	7	42711	0.918592	**0.240451**	0.243461	0.968942
L7_3	6	30135	0.960985	**0.356182**	0.356860	0.968619
L7_4	13	22608	**0.097177**	0.379325	0.381078	0.832390
L7_5	13	8344	0.820768	**0.198944**	0.203315	0.961631
L7_6	13	12896	0.827021	**0.053005**	0.055044	0.974603
L7_7	13	3473	0.912125	**0.054519**	0.059149	0.928100
L7_8	13	28476	0.804303	**0.231228**	0.231309	0.985363
Snort24	24	13882	**0.037515**	0.103243	0.108468	0.957364
Snort31	31	19522	**0.053581**	0.058584	0.061309	0.915806
Snort34	34	13834	**0.032259**	0.058067	0.060473	0.947866
BRO217	217	6533	0.061814	**0.035062**	0.224820	0.514522
Synthetic_1	50	248	0.111281	0.011656	0.186697	**0.007749**
Synthetic_2	10	78337	0.099659	**0.026233**	0.030254	0.998601
Synthetic_3	50	8338	0.948123	**0.014934**	0.018575	0.335646
Synthetic_4	10	5290	0.990808	**0.042752**	0.046357	0.958690
Synthetic_5	50	7828	0.947048	**0.016112**	0.019762	0.326956
Synthetic_6	50	14496	0.973929	**0.048839**	0.173284	0.478337

4.2 Searching Time Comparison

This section compares the searching time of our method CRD with that of the original DFA and the δFA. We generate a random text of size 10MB to search against with the synthetic rule sets.

Both the δFA and our method need to store a residual sparse matrix using compact data structure. To represent the sparse matrix, we store the nonempty elements in each row in a sorted array, and accessing an element is accomplished by doing binary searching on it. To avoid unnecessary probes into the sparse table, we use the bloom filter [15] technique to indicate whether a position in the sparse matrix is empty or not (See Algorithm 2). This simple but efficient trick eliminates most of the probes into the sparse matrix.

Searching time of the algorithms on synthetic rule sets is listed in table 2. Despite its high compressibility, our method CRD still runs at fast speed comparable to that of the original DFA. The searching time increase of our method is limited within 20%∼25%. The δFA method, which is designed for hardware

Table 2. Searching time (in seconds) of the algorithms on synthetic rule sets

Rule set	Original DFA	CRD	δFA
Synthetic_1	0.1250	0.1516	103.813
Synthetic_2	0.1218	0.1485	48.094
Synthetic_3	0.1204	0.1500	211.734
Synthetic_4	0.1204	0.1500	224.672
Synthetic_5	0.1203	0.1484	188.937
Synthetic_6	0.1250	0.1500	200.735

implementation, is significantly slower than the original DFA and our method. State switching in δFA costs $O(|\Sigma|)$ resulting in poor performance.

5 Conclusion

The huge memory usage of regular expressions' DFA prevents it from being applied on large rule sets. To deal with this problem, we proposed a matrix decomposition-based method for DFA table compression. The basic idea of our method is to decompose a DFA table into the sum of a row vector, a column vector and a sparse matrix, all of which cost very little space. Experiments on typical rule sets show that the proposed method significantly reduces the memory usage and still runs at fast searching speed.

Acknowledgment

This work is supported by the National Basic Research Program of China (973) under grant No. 2007CB311100 and the National Information Security Research Program of China (242) under grant No. 2009A43. We would like to thank Michela Becchi for providing her useful *regex-tool* to us for evaluation. We are also grateful to the anonymous referees for their insightful comments.

References

1. Thompson, K.: Programming Techniques: Regular expression search algorithm. Communications of the ACM 11(6), 419–422 (1968)
2. Myers, E.W.: A four Russians algorithm for regular expression pattern matching. Journal of the ACM 39(2), 430–448 (1992)
3. Baeza-Yates, R.A., Gonnet, G.H.: Fast text searching for regular expressions or automaton searching on tries. Journal of the ACM 43(6), 915–936 (1996)
4. Navarro, G., Raffinot, M.: Compact DFA representation for fast regular expression search. In: Brodal, G.S., Frigioni, D., Marchetti-Spaccamela, A. (eds.) WAE 2001. LNCS, vol. 2141, pp. 1–12. Springer, Heidelberg (2001)
5. Navarro, G., Raffinot, M.: Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In: Proceedings of the 5th Annual International Conference on Computational Molecular Biology, pp. 231–240 (2001)

6. Champarnaud, J.-M., Coulon, F., Paranthoen, T.: Compact and Fast Algorithms for Regular Expression Search. Intern. J. of Computer. Math. 81(4) (2004)
7. Yu, F., Chen, Z., Diao, Y.: Fast and memory-efficient regular expression matching for deep packet inspection. In: Proceedings of the 2006 ACM/IEEE symposium on Architecture for Networking and Communications Systems, pp. 93–102 (2006)
8. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. ACM SIGCOMM Computer Communication Review 36(4), 339–350 (2006)
9. Becchi, M., Crowley, P.: An improved algorithm to accelerate regular expression evaluation. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems, pp. 145–154 (2007)
10. Ficara, D., Giordano, S., Procissi, G., Vitucci, F., Antichi, G., Pietro, A.D.: An improved DFA for fast regular expression matching. ACM SIGCOMM Computer Communication Review 38(5), 29–40 (2008)
11. Smith, R., Estan, C., Jha, S.: XFA: Faster signature matching with extended automata. In: IEEE Symposium on Security and Privacy, Oakland, pp. 187–201 (May 2008)
12. Kumar, S., Chandrasekaran, B., Turner, J., Varghese, G.: Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems, pp. 155–164 (2007)
13. Becchi, M., Cadambi, S.: Memory-efficient regular expression search using state merging. In: 26th IEEE International Conference on Computer Communications, pp. 1064–1072 (2007)
14. Majumder, A., Rastogi, R., Vanama, S.: Scalable regular expression matching on data streams. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, Canada, pp. 161–172 (2008)
15. Bloom, B.H.: Spacetime Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM 13(7), 422–426 (1970)
16. http://l7-filter.sourceforge.net/
17. http://www.snort.org/
18. http://www.bro-ids.org/
19. http://regex.wustl.edu/

Relational String Verification Using Multi-track Automata*

Fang Yu[1], Tevfik Bultan[2], and Oscar H. Ibarra[2]

[1] National Chengchi University, Taipei, Taiwan
yuf@nccu.edu.tw
[2] University of California, Santa Barbara, CA, USA
{bultan,ibarra}@cs.ucsb.edu

Abstract. Verification of string manipulation operations is a crucial problem in computer security. In this paper, we present a new relational string verification technique based on multi-track automata. Our approach is capable of verifying properties that depend on relations among string variables. This enables us to prove that vulnerabilities that result from improper string manipulation do not exist in a given program. Our main contributions in this paper can be summarized as follows: (1) We formally characterize the string verification problem as the reachability analysis of *string systems* and show decidability/undecidability results for several string analysis problems. (2) We develop a sound symbolic analysis technique for string verification that over-approximates the reachable states of a given string system using multi-track automata and summarization. (3) We evaluate the presented techniques with respect to several string analysis benchmarks extracted from real web applications.

1 Introduction

The most important Web application vulnerabilities are due to inadequate manipulation of string variables [10]. In this paper we investigate the *string verification problem:* Given a program that manipulates strings, we want to verify assertions about string variables. For example, we may want to check that at a certain program point a string variable cannot contain a specific set of characters. This type of checks can be used to prevent SQL injection attacks where a malicious user includes special characters in the input string to inject unintended commands to the queries that the Web application constructs (using the input provided by the user) and sends to a backend database. As another example, we may want to check that at a certain program point a string variable should be prefix or suffix of another string variable. This type of checks can be used to prevent Malicious File Execution (MFE) attacks where Web application developers concatenate potentially hostile user input with file functions that lead to inclusion or execution of untrusted files by the Web server.

We formalize the string verification problem as reachability analysis of *string systems* (Section 2). After demonstrating that the string analysis problem is undecidable

* This work is supported by NSF grants CCF-0916112, CCF-0716095, and CCF-0524136.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 290–299, 2011.

in general, we present and implement a forward symbolic reachability analysis technique that computes an over-approximation of the reachable states of a string system using widening and summarization (Section 4). We use multi-track deterministic finite automata (DFAs) as a symbolic representation to encode the set of possible values that string variables can take at a given program point. Unlike prior string analysis techniques, our analysis is *relational*, i.e., it is able to keep track of the relationships among the string variables, improving the precision of the string analysis and enabling verification of invariants such as $X_1 = X_2$ where X_1 and X_2 are string variables. We develop the precise construction of multi-track DFAs for linear word equations, such as $c_1 X_1 c_2 = c_1' X_2 c_2'$ and show that non-linear word equations (such as $X_1 = X_2 X_3$) cannot be characterized precisely as a multi-track DFA (Section 3). We propose a regular approximation for non-linear equations and show how these constructions can be used to compute the post-condition of branch conditions and assignment statements that involve concatenation. We use summarization for inter-procedural analysis by generating a multi-track automaton (transducer) characterizing the relationship between the input parameters and the return values of each procedure (Section 4). To be able to use procedure summaries during our reachability analysis we *align* multi-track automata so that normalized automata are closed under intersection. We implemented these algorithms using the MONA automata package [5] and analyzed several PHP programs demonstrating the effectiveness of our string analysis techniques (Section 5).

Related Work. The use of automata as a symbolic representation for verification has been investigated in other contexts [4]. In this paper, we focus on verification of string manipulation operations, which is essential to detect and prevent crucial web vulnerabilities. Due to its importance in security, string analysis has been widely studied. One influential approach has been grammar-based string analysis that statically computes an over-approximation of the values of string expressions in Java programs [6] which has also been used to check for various types of errors in Web applications [8, 9, 12]. In [9, 12], multi-track DFAs, known as *transducers*, are used to model replacement operations. There are also several recent string analysis tools that use symbolic string analysis based on DFA encodings [7, 11, 14, 15]. Some of them are based on symbolic execution and use a DFA representation to model and verify the string manipulation operations in Java programs [7, 11]. In our earlier work, we have used a DFA based symbolic reachability analysis to verify the correctness of string sanitization operations in PHP programs [13–15]. Unlike the approach we proposed in this paper, all of the results mentioned above use single track DFA and encode the reachable configurations of each string variable separately. Our multi-track automata encoding not only improves the precision of the string analysis but also enables verification of properties that cannot be verified with the previous approaches. We have also investigated the boundary of decidability for the string verification problem. Bjørner et al. [2] show the undecidability result with replacement operation. In this paper we consider only concatenation and show that string verification problem is undecidable even for deterministic string systems with only three unary string variables and non-deterministic string systems with only two string variables if the comparison of two variables are allowed.

prog ::= *decl** *func**
decl ::= `decl` *id*$^+$;
func ::= *id* (*id**) `begin` *decl** *lstmt*$^+$ `end`
lstmt ::= *l*:*stmt*
stmt ::= *seqstmt* | `if` *exp* `then goto` *l*; | `goto` *L*; where *L* is a set of labels
| `input` *id*; | `output` *exp*; | `assert` *exp*;
seqstmt::=*id* := *sexp*; | *id* := `call` *id* (*sexp**);
exp ::= *bexp* | *exp* ∧ *exp* | ¬ *exp*
bexp ::= *atom* = *sexp*
sexp ::= *sexp*.*atom* | *atom* | *suffix*(*id*) | *prefix*(*id*)
atom ::= *id* | *c*, where *c* is a string constant

Fig. 1. The syntax of string systems

2 String Systems

We define the syntax of string systems in Figure 1. We only consider string variables and hence variable declarations need not specify a type. All statements are labeled. We only consider one string operation (concatenation) in our formal model; however, our symbolic string analysis techniques can be extended to handle complex string operations (such as replacement [14]). Function calls use call-by-value parameter passing. We allow goto statements to be non-deterministic (if a goto statement has multiple target labels, then one of them is chosen non-deterministically). If a string system contains a non-deterministic goto statement it is called a non-deterministic string system, otherwise, it is called a deterministic string system.

There are several attributes we can use to classify string systems such as deterministic (D) or non-deterministic (N) string systems, the number of variables in the string systems, and the alphabet used by the string variables, e.g., unary (U), binary (B), or arbitrary (K) alphabet. Finally, we can restrict the set of string expressions that can be used in the assignment and conditional branch instructions. As an instance, $NB(X_1, X_2)^{X_i:=X_ic}_{X_1=X_2}$ denotes a non-deterministic string system with a binary alphabet and two string variables (X_1 and X_2) where variables can only concatenate constant strings from the right and compared to each other. We use a to denote a single symbol, and c, d to denote constant strings. $c = \mathit{prefix}(X_i)$ evaluates to true if c is a prefix of X_i, and $c = \mathit{suffix}(X_i)$ evaluates to true if c is a suffix of X_i. We define the *reachability problem for string systems* is the problem of deciding, given a string system and a configuration (i.e., the instruction label and the values of the variables), whether at some point during a computation, the configuration will be reached. We have the following results:

Theorem 1. *The reachability problem for:*

1. $NB(X_1, X_2)^{X_i:=X_ic}_{X_1=X_2}$ *is undecidable,*
2. $DU(X_1, X_2, X_3)^{X_i:=X_ic}_{X_1=X_3, X_2=X_3}$ *is undecidable,*
3. $DU(X_1, X_2, X_3, X_4)^{X_i:=X_ic}_{X_1=X_3, X_2=X_4}$ *is undecidable,*
4. $NU(X_1, X_2)^{X_i:=X_ic}_{X_1=X_2, c=X_i, c=\mathit{prefix}(X_i), c=\mathit{suffix}(X_i)}$ *is decidable,*

5. $NK(X_1, X_2, \ldots, X_k)^{X_i:=dX_ic}_{c=X_i, c=prefix(X_i), c=suffix(X_i)}$ *is decidable, and*
6. $DK(X_1, X_2, , \ldots, X_k)^{X_i:=X_ia, X_i:=aX_i}_{X_1=X_2, c=X_i, c=prefix(X_i), c=suffix(X_i)}$ *is decidable.*

Theorem 1 demonstrates the complexity boundaries for verification of string systems. Theorem 1.1, 1.2 and 1.3 show that the string verification problem can be undecidable even when we restrict a non-deterministic string system to two binary variables, or a deterministic string system to three unary variables or four unary variables with specific comparisons. Theorem 1.4 shows that the three variables in Theorem 1.2 are necessary in the sense that when there are only two variables, reachability is decidable, even when the string system is nondeterministic. Theorem 1.5 and 1.6, on the other hand, demonstrate that there are non-trivial string verification problems that are decidable. Since the general string verification problem is undecidable, it is necessary to develop conservative approximation techniques for verification of string systems. In this paper we propose a sound string verification approach based on symbolic reachability analysis with conservative approximations where multi-track automata are used as a symbolic representation. Some examples of string systems that can be verified using our analysis are given in [16].

3 Regular Approximation of Word Equations

Our string analysis is based on the following observations: (1) The transitions and the configurations of a string system can be symbolically represented using word equations with existential quantification, (2) word equations can be represented/approximated using multi-track DFAs, which are closed under intersection, union, complement, and projection, and (3) the operations required during reachability analysis (such as equivalence checking) can be computed on DFAs.

Multi-track DFAs. A multi-track DFA is a DFA but over the alphabet that consists of many tracks. An n-track alphabet is defined as $(\Sigma \cup \{\lambda\})^n$, where $\lambda \notin \Sigma$ is a special symbol for padding. We use $w[i]$ $(1 \leq i \leq n)$ to denote the i^{th} track of $w \in (\Sigma \cup \{\lambda\})^n$. An *aligned* multi-track DFA is a multi-track DFA where all tracks are left justified (i.e., λ's are right justified). That is, if w is accepted by an aligned n-track DFA M, then for $1 \leq i \leq n$, $w[i] \in \Sigma^*\lambda^*$. We also use $\hat{w}[i] \in \Sigma^*$ to denote the longest λ-free prefix of $w[i]$. It is clear that aligned multi-track DFA languages are closed under intersection, union, and homomorphism. Let M_u be the aligned n-track DFA that accepts the (aligned) universe, i.e., $\{w \mid \forall i.w[i] \in \Sigma^*\lambda^*\}$. The complement of the language accepted by an aligned n-track DFA M is defined by *complement modulo alignment*, i.e., the intersection of the complement of $L(M)$ with $L(M_u)$. For the following descriptions, a multi-track DFA is an aligned multi-track DFA unless we explicitly state otherwise.

Word Equations. A word equation is an equality relation of two words that concatenate variables from a finite set $\mathbf{X}$ and words from a finite set of constants $\mathcal{C}$. The general form of word equations is defined as $v_1 \ldots v_n = v'_1 \ldots v'_m$, where $\forall i, v_i, v'_i \in \mathbf{X} \cup \mathcal{C}$. The following theorem identifies the basic forms of word equations. For example, a word equation $f : X_1 = X_2dX_3X_4$ is equivalent to $\exists X_{k_1}, X_{k_2}.X_1 = X_2X_{k_1} \wedge X_{k_1} = dX_{k_2} \wedge X_{k_2} = X_3X_4$.

Theorem 2. *Word equations and Boolean combinations ($\neg$, $\wedge$ and $\vee$) of these equations can be expressed using equations of the form $X_1 = X_2c$, $X_1 = cX_2$, $c = X_1X_2$, $X_1 = X_2X_3$, Boolean combinations of such equations and existential quantification.*

Let f be a word equation over $\mathbf{X}$= $\{X_1, X_2, \ldots, X_n\}$ and $f[c/X]$ denote a new equation where X is replaced with c for all X that appears in f. We say that an n-track DFA M *under-approximates* f if for all $w \in L(M)$, $f[\hat{w}[1]/X_1, \ldots, \hat{w}[n]/X_n]$ holds. We say that an n-track DFA M *over-approximates* f if for any $s_1, \ldots, s_n \in \Sigma^*$ where $f[s_1/X_1, \ldots, s_n/X_n]$ holds, there exists $w \in L(M)$ such that for all $1 \leq i \leq n, \hat{w}[i] = s_i$. We call M *precise* with respect to f if M both under-approximates and over-approximates f.

Definition 1. *A word equation f is regularly expressible if and only if there exists a multi-track DFA M such that M is precise with respect to f.*

Theorem 3.
1. *$X_1 = X_2c$, $X_1 = cX_2$, and $c = X_1X_2$ are regularly expressible, as well as their Boolean combinations.*
2. *$X_1 = cX_2$ is regularly expressible but the corresponding M has exponential number of states in the length of c.*
3. *$X_1 = X_2X_3$ is not regularly expressible.*

We are able to compute multi-track DFAs that are precise with respect to word equations: $X_1 = X_2c$, $X_1 = cX_2$, and $c = X_1X_2$. Since $X_1 = X_2X_3$ is not regularly expressible, below, we describe how to compute DFAs that approximate such non-linear word equations. Using the DFA constructions for these four basic forms we can construct multi-track DFAs for all word equations and their Boolean combinations (if the word equation contains a non-linear term then the constructed DFA will approximate the equation, otherwise it will be precise). The Boolean operations conjunction, disjunction and negation can be handled with intersection, union, and complement modulo alignment of the multi-track DFAs, respectively. Existential quantification on the other hand, can be handled using homomorphism, where given a word equation f and a multi-track automaton M such that M is precise with respect to f, then the multi-track automaton $M \ \lfloor_i$ is precise with respect to $\exists X_i.f$ where $M \ \lfloor_i$ denotes the result of erasing the i^{th} track (by homomorphism) of M.

Construction of $X_1 = X_2X_3$. Since Theorem 3 shows that $X_1 = X_2X_3$ is not regularly expressible, it is necessary to construct a conservative (*over* or *under*) approximation of $X_1 = X_2X_3$. We first propose an *over* approximation construction for $X_1 = X_2X_3$. Let $M_1 = \langle Q_1, \Sigma, \delta_1, I_1, F_1\rangle$, $M_2 = \langle Q_2, \Sigma, \delta_2, I_2, F_2\rangle$, and $M_3 = \langle Q_3, \Sigma, \delta_3, I_3, F_3\rangle$ accept values of X_1, X_2, and X_3, respectively. $M = \langle Q, (\Sigma \cup \{\lambda\})^3, \delta, I, F\rangle$ is constructed as follows.

- $Q \subseteq Q_1 \times Q_2 \times Q_3 \times Q_3$,
- $I = (I_1, I_2, I_3, I_3)$,
- $\forall a, b \in \Sigma, \delta((r, p, s, s'), (a, a, b)) = (\delta_1(r, a), \delta_2(p, a), \delta_3(s, b), s')$,
- $\forall a, b \in \Sigma, p \in F_2, s \notin F_3, \delta((r, p, s, s'), (a, \lambda, b)) = (\delta_1(r, a), p, \delta_3(s, b), \delta_3(s', a))$,
- $\forall a \in \Sigma, p \in F_2, s \in F_3, \delta((r, p, s, s'), (a, \lambda, \lambda)) = (\delta_1(r, a), p, s, \delta_3(s', a))$,
- $\forall a \in \Sigma, p \notin F_2, s \in F_3, \delta((r, p, s, s'), (a, a, \lambda)) = (\delta_1(r, a), \delta_2(p, a), s, s')$,
- $F = \{(r, p, s, s') \mid r \in F_1, p \in F_2, s \in F_3, s' \in F_3\}$.

The intuition is as follows: M traces M_1, M_2 and M_3 on the first, second and third tracks, respectively, and makes sure that the first and second tracks match each other. After reaching an accepting state in M_2, M enforces the second track to be λ and starts to trace M_3 on the first track to ensure the rest (suffix) is accepted by M_3. $|Q|$ is $O(|Q_1| \times |Q_2| \times |Q_3| + |Q_1| \times |Q_3| \times |Q_3|)$. For all $w \in L(M)$, the following hold:

- $\hat{w}[1] \in L(M_1), \hat{w}[2] \in L(M_2), \hat{w}[3] \in L(M_3)$,
- $\hat{w}[1] = \hat{w}[2]w'$ and $w' \in L(M_3)$,

Note that w' may not be equal to $\hat{w}[3]$, i.e., there exists $w \in L(M)$, $\hat{w}[1] \neq \hat{w}[2]\hat{w}[3]$, and hence M is not precise with respect to $X_1 = X_2X_3$. On the other hand, for any w such that $\hat{w}[1] = \hat{w}[2]\hat{w}[3]$, we have $w \in L(M)$, hence M is a regular *over*-approximation of $X_1 = X_2X_3$.

Below, we describe how to construct a regular *under*-approximation of $X_1 = X_2X_3$ (which is necessary for conservative approximation of its complement set). We use the idea that if $L(M_2)$ is a finite set language, one can construct the DFA M that satisfies $X_1 = X_2X_3$ by explicitly taking the union of the construction of $X_1 = cX_3$ for all $c \in L(M_2)$. If $L(M_2)$ is an infinite set language, we construct a regular *under*-approximation of $X_1 = X_2X_3$ by considering a (finite) subset of $L(M_2)$ where the length is bounded. Formally speaking, for each $k \geq 0$ we can construct M_k, so that $w \in L(M_k), \hat{w}[1] = \hat{w}[2]\hat{w}[3]$, $\hat{w}[1] \in L(M_1)$, $\hat{w}[3] \in L(M_3)$, $\hat{w}[2] \in L(M_2)$ and $|\hat{w}[2]| \leq k$. It follows that M_k is a regular *under*-approximation of $X_1 = X_2X_3$. If $L(M_2)$ is a finite set language, there exists k (the length of the longest accepted word) where $L(M_k)$ is precise with respect to $X_1 = X_2X_3$. If $L(M_2)$ is an infinite set language, there does not exist such k so that $L(M_k)$ is precise with respect to $X_1 = X_2X_3$, as we have proven non-regularity of $X_1 = X_2X_3$.

4 Symbolic Reachability Analysis

Our symbolic reachability analysis involves two main steps: forward fixpoint computation and summarization.

Forward Fixpoint Computation. The first phase of our analysis is a standard forward fixpoint computation on multi-track DFAs. Each program point is associated with a single multi-track DFA, where each track is associated with a single string variable $X \in \mathbf{X}$. We use $M[l]$ to denote the multi-track automaton at the program label l. The forward fixpoint computation algorithm is a standard work-queue algorithm. Initially, for all labels l, $L(M[l]) = \emptyset$. We iteratively compute the post-images of the statements and join the results to the corresponding automata. For a $stmt$ in the form: $X := sexp$, the post-image is computed as:

$$\textsc{post}(M, stmt) \equiv (\exists X.M \cap \textsc{Construct}(X' = sexp, +))[X/X'].$$

$\textsc{Construct}(exp, b)$ returns the DFA that accepts a regular approximation of exp, where $b \in \{+, -\}$ indicates the direction (*over* or *under*, respectively) of approximation if needed. During the construction, we recursively push the negations ($\neg$) (and flip the direction) inside to the basic expressions ($bexp$), and use the corresponding construction

```
f(X)
begin
1: goto 2, 3;
2: X: = call f(X.a);
3: return X;
end
```

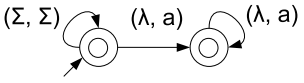


Fig. 2. A function and its summary DFA

of multi-track DFAs discussed in the previous section. We use function summaries to handle function calls. Each function f is summarized as a finite state transducer, denoted as M_f, which captures the relations among input variables (parameters), denoted as X_p, and return values. The return values are tracked in the output track, denoted as X_o. We discuss the generation of the transducer M_f below. For a $stmt$ in the form X:= `call` $f(e_1, \ldots, e_n)$, the post-image is computed as:

$$\text{POST}(M, stmt) \equiv (\exists X, X_{p_1}, \ldots X_{p_n}.M \cap M_I \cap M_f)[X/X_o],$$

where M_I = CONSTRUCT($\bigwedge_i X_{p_i} = e_i$, +). The process terminates when we reach a fixpoint. To accelerate the fixpoint computation, we extend our automata widening operator [14], denoted as ∇, to multi-track automata. We identify equivalence classes according to specific equivalence conditions and merge states in the same equivalence class [1, 3]. The following lemma shows that the equality relations among tracks are preserved while widening multi-track automata.

Lemma 1. *if* $L(M) \subseteq L(x = y)$ *and* $L(M') \subseteq L(x = y)$, $L(M \nabla M') \subseteq L(x = y)$.

Summarization. We compute procedure summaries in order to handle procedure calls. We assume parameter-passing with call-by-value semantics and we are able to handle recursion. Each function f is summarized as a multi-track DFA, denoted as M_f, that captures the relation among its input variables and return values.

Consider the recursive function f shown in Figure 2 with one parameter. f nondeterministically returns its input (goto 3) or makes a self call (goto 2) by concatenating its input and a constant a. The generated summary for this function is also shown in Figure 2. M_f is a 2-track DFA, where the first track is associated with its parameter X_{p_1}, and the second track is associated with X_o representing the return values. The edge (Σ, Σ) represents a set of identity edges. In other words, $\delta(q, (\Sigma, \Sigma)) = q'$ means $\forall a \in \Sigma, \delta(q, (a, a)) = q'$. The summary DFA M_f precisely captures the relation $X_o = X_{p_1}.a^*$ between the input variable and the return values.

During the summarization phase, (possibly recursive) functions are summarized as unaligned multi-track DFAs that specify the relations among their inputs and return values. We first build (possibly cyclic) dependency graphs to specify how the inputs flow to the return values. Each node in the dependency graph is associated with an unaligned multi-track DFA that traces the relation among inputs and the value of that node. An unaligned multi-track DFA is a multi-track DFA where λs might not be right justified. Return values of a function are represented with an auxiliary output track. Given a function f with n parameters, M_f is an unaligned $(n+1)$-track DFA, where n tracks represent the n input parameters and one track X_o is the output track representing

the return values. We iteratively compute post images of reachable relations and join the results until we reach a fixpoint. Upon termination, the summary is the union of the unaligned DFAs associated with the return nodes. To compose these summaries at the call site, we also propose an alignment algorithm to align (so that λ's are right justified) an unaligned multi-track DFA.

Once the summary DFA M_f has been computed, it is not necessary to reanalyze the body of f. To compute the post-image of a call to f we intersect the values of input parameters with M_f and use existential quantification to obtain the return values. Let M be a one-track DFA associated with X where $L(M) = \{b\}$. $\textsc{post}(M, X := \texttt{call}\ f(X))$ returns M' where $L(M') = ba^*$ for the example function shown above. As another example, let M be a 2-track DFA associated with X, Y that is precise with respect to $X = Y$. Then $\textsc{post}(M, X := \texttt{call}\ f(X))$ returns M' which is precise with respect to $X = Y.a^*$ precisely capturing the relation between X and Y after the execution of the function call. As discussed above, M' is computed by $(\exists X, X_{p_1}.M \cap M_I \cap M_f)[X/X_o]$, where $L(M_I) = \textsc{Construct}(X_{p_1} = X, +)$.

5 Experiments

We evaluate our approach against three kinds of benchmarks: 1) Basic benchmarks, 2) XSS/SQLI benchmarks, and 3) MFE benchmarks. These benchmarks represent typical string manipulating programs along with string properties that address severe web vulnerabilities.

In the first set, we demonstrate that our approach can prove implicit equality properties of string systems. We wrote two small programs. CheckBranch (B1) has if branch ($X_1 = X_2$) and else branch ($X_1 \neq X_2$). In the else branch, we assign a constant string c to X_1 and then assign the same constant string to X_2. We check at the merge point whether $X_1 = X_2$. In CheckLoop (B2) we assign variables X_1 and X_2 the same constant string at the beginning, and iteratively append another constant string to both in an infinite loop. We check whether $X_1 = X_2$ at the loop exit. Let M accept the values of X_1 and X_2 upon termination. The equality assertion holds when $L(M) \subseteq L(M_a)$, where M_a is $\textsc{Construct}(X_1 = X_2, -)$. We use "$-$" to construct (under approximation) automata for assertions to ensure the soundness of our analysis. Using multi-track DFAs, we prove the equality property (result "true") whereas we are unable to prove it using single-track DFAs (result "false") as shown in Table 1 (B1 and B2). Though these benchmark examples are simple, to the best of our knowledge, there are no other string analysis tools that can prove equality properties in these benchmarks.

In the second set, we check existence of Cross-Site Scripting (XSS) and SQL Injection (SQLI) vulnerabilities in Web applications with known vulnerabilities. We check whether at a specific program point, a sensitive function may take an attack string as its input. If so, we say that the program is vulnerable (result "vul") with respect to the given attack pattern. To identify XSS/SQLI attacks, we check intersection emptiness against all possible input values that reach a sensitive function at a given program point and the attack strings specified as a regular language. Though one can check such vulnerabilities using single-track DFAs [14], using multi-track automata, we can precisely interpret branch conditions, such as `$www=$url`, that cannot be precisely expressed

Table 1. Experimental results. DFA(s): the minimized DFA(s) associated with the checked program point. state: number of states. bdd: number of bdd nodes. Benchmark: Application, script (line number). S1: MyEasyMarket-4.1, trans.php (218). S2: PBLguestbook-1.32, pblguestbook.php (1210), S3:Aphpkb-0.71, saa.php (87), and S4: BloggIT 1.0, admin.php (23). M1: PBLguestbook-1.32, pblguestbook.php (536). M2: MyEasyMarket-4.1, prod.php (94). M3: MyEasyMarket-4.1, prod.php (189). M4: php-fusion-6.01, db_backup.php (111). M5: php-fusion-6.01, forums_prune.php (28).

	Single-track				Multi-track			
Ben	Result	DFAs/ Composed DFA state(bdd)	Time user+sys(sec)	Mem (kb)	Result	DFA state(bdd)	Time user+sys(sec)	Mem (kb)
B1	false	15(107), 15(107) /33(477)	0.027 + 0.006	410	true	14(193)	0.070 + 0.009	918
B2	false	6(40), 6(40) / 9(120)	0.022+0.008	484	true	5(60)	0.025+0.006	293
S1	vul	2(20), 9(64), 17(148)	0.010+0.002	444	vul	65(1629)	0.195+0.150	1231
S2	vul	9(65), 42(376)	0.017+0.003	626	vul	49(1205)	0.059+0.006	4232
S3	vul	11(106), 27(226)	0.032+0.003	838	vul	47(2714)	0.153+0.008	2684
S4	vul	53(423), 79(633)	0.062+0.005	1696	vul	79(1900)	0.226+0.003	2826
M1	vul	2(8), 28(208) / 56(801)	0.027+0.003	621	no	50(3551)	0.059+0.002	1294
M2	vul	2(20), 11(89) / 22(495)	0.013+0.004	555	no	21(604)	0.040+0.004	996
M3	vul	2(20), 2(20) / 5(113)	0.008+0.002	417	no	3(276)	0.018+0.001	465
M4	vul	24(181), 2(8), 25(188) / 1201(25949)	0.226+0.025	9495	no	181(9893)	0.784+0.07	19322
M5	vul	2(8), 14(101), 15(108) / 211(3195)	0.049+0.008	1676	no	62(2423)	0.097+0.005	1756

using single-track automata, and obtain more accurate characterization of inputs of the sensitive functions. For the vulnerabilities identified in these benchmarks (S1 to S4), we did not observe false alarms that result from the approximation of the branch conditions.

The last set of benchmarks show that the precision that is obtained using multi-track DFAs can help us in removing false alarms generated by single-track automata based string analysis. These benchmarks represent *malicious file execution* (MFE) attacks. Such vulnerabilities are caused because developers directly use or concatenate potentially hostile input with file or stream functions, or improperly trust input files. We systematically searched web applications for program points that execute file functions, such as `include` and `fopen`, whose arguments may be influenced by external inputs. At these program points, we check whether the retrieved files and the external inputs are consistent with what the developers intend. We manually generate a multi-track DFA M_{vul} that accepts a set of possible violations for each benchmark, and apply our analysis on the sliced program segments. Upon termination, we report that the file function is vulnerable (result "vul") if $L(M) \cap L(M_{vul}) \neq \emptyset$. M is the composed DFA of the listed single-track DFAs in the single-track analysis. As shown in Table 1 (M1 to M5), using multi-track DFAs we are able to verify that MFE vulnerabilities do not exist (result "no") whereas string analysis using single-track DFAs raises false alarms for all these examples.

We have shown that multi-track DFAs can handle problems that cannot be handled by multiple single-track DFAs, but at the same time, they use more time and memory. For these benchmarks, the cost seems affordable. As shown in Table 1, in all tests, the multi-track DFAs that we computed (even for the composed ones) are smaller than the product of the corresponding single-track DFAs. One advantage of our implementation is symbolic DFA representation (provided by the MONA DFA library [5]), in which transition relations of the DFA are represented as Multi-terminal Binary Decision Diagrams (MBDDs). Using the symbolic DFA representation we avoid the potential

exponential blow-up that can be caused by the product alphabet. However, in the worst case the size of the MBDD can still be exponential in the number of tracks.

6 Conclusion

In this paper, we presented a formal characterization of the string verification problem, investigated the decidability boundary for string systems, and presented a novel verification technique for string systems. Our verification technique is based on forward symbolic reachability analysis with multi-track automata, conservative approximations of word equations and summarization. We demonstrated the effectiveness of our approach on several benchmarks.

References

1. Bartzis, C., Bultan, T.: Widening arithmetic automata. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 321–333. Springer, Heidelberg (2004)
2. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009)
3. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
4. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
5. BRICS. The MONA project, http://www.brics.dk/mona/
6. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
7. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A static analysis framework for detecting sql injection vulnerabilities. In: COMPSAC, pp. 87–96 (2007)
8. Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. In: ICSE, pp. 645–654 (2004)
9. Minamide, Y.: Static approximation of dynamically generated web pages. In: WWW 2005, pp. 432–441 (2005)
10. Open Web Application Security Project (OWASP). Top ten project (May 2007), http://www.owasp.org/
11. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: TAICPART-MUTATION, DC, USA, pp. 13–22 (2007)
12. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: ICSE, pp. 171–180 (2008)
13. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for php. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010)
14. Yu, F., Bultan, T., Cova, M., Ibarra, O.H.: Symbolic string verification: An automata-based approach. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 306–324. Springer, Heidelberg (2008)
15. Yu, F., Bultan, T., Ibarra, O.H.: Symbolic string verification: Combining string analysis and size analysis. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 322–336. Springer, Heidelberg (2009)
16. Yu, F., Bultan, T., Ibarra, O.H.: Verification of string manipulating programs using multi-track automata. Technical Report 2009-14, Computer Science Department, University of California, Santa Barbara (August 2009)

A Note on a Tree-Based 2D Indexing*

Jan Žďárek and Bořivoj Melichar

Department of Theoretical Computer Science,
Faculty of Information Technology, Czech Technical University in Prague,
Kolejní 550/2, 160 00 Praha 6, Czech Republic
{melichar,zdarekj}@fit.cvut.cz

Abstract. A new approach to the 2D pattern matching and specifically to 2D text indexing is proposed. We present the transformation of 2D structures into the form of a tree, preserving the context of each element of the structure. The tree can be linearised using the prefix notation into the form of a text (a string) and we do the pattern matching in this text. Over this representation pushdown automata indexing the 2D text are constructed. They allow to search for 2D prefixes, suffixes, or factors of the 2D text in time proportional to the size of the representation of a 2D pattern. This result achieves the properties analogous to the results obtained in tree pattern matching and string indexing.

1 Introduction

In the area of string matching, there are many methods to preprocess the text to obtain some kind of its index. Then the pattern is to be searched in the index. Text indexing is commonly used in situation where many patterns shall be tested against a fixed text (e.g. in biological or other database oriented applications).

This paper presents pushdown automata that serve as a pattern matching and indexing tool for linearised 2D text (pictures). We present the transformation of 2D structures into the form of a tree, preserving the context of each element of the structure. The tree can be linearised using the prefix notation into the form of a string and we do the pattern matching in this text. Pushdown automata indexing the 2D text allow to search for 2D prefixes, suffixes, or factors of the 2D text in time proportional to the size of the 2D pattern representation. This result achieves the properties analogous to the results obtained in tree pattern matching and string indexing.

Organisation

First, we present a short list of useful notions and notations and an overview of the previous works. Then we present a *vertex splitting* method for representation of a 2D text and in the following chapter the matching in such a representation is described. The next chapter shows construction methods of 2D indexing pushdown automata for 2D prefix, suffix and factor matching. The last chapter is the conclusion. All proofs and some examples, removed due to strict space limitations, can be found in [19, Chapter 5].

* Partially supported by Czech Science Foundation project No. 201/09/0807, and by MŠMT of the Czech Republic under research program No. MSM 6840770014.

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 300–309, 2011.

2 Basic Notions

Let A be a finite alphabet and its elements be called symbols. A set of strings over A is denoted by A^*. The empty string is denoted by ε. Language L is any subset of A^*, $L \subseteq A^*$.

A picture (2D string) is a rectangular arrangement of symbols taken from a finite alphabet. The set of all pictures over alphabet A is denoted by A^{**} and a 2D language over A is thus any subset of A^{**}. The set of all pictures of size $(n \times n')$ over A, where $n, n' > 0$, is denoted by $A^{(n \times n')}$. Let the empty picture of size (0×0) be denoted by λ. (Giammarresi and Restivo [1] discuss the theory of 2D languages in detail.) The size of a picture, say P, is the size of its rectangular shape, denoted by $|P|$ or $(x \times y)$, and its numerical value is the product of its $\boldsymbol{x}$ and $\boldsymbol{y}$ components, $|P| = xy$. The origin of P is the element $P[1, 1]$.

Let $R \in A^{(n \times n')}$ be a picture of size $(n \times n')$ and $P \in A^{(m \times m')}$ be a picture of size $(m \times m')$. P is a sub-picture of R if and only if $m \leq n \wedge m' \leq n'$ ($|P| \leq |R|$) and every element of P occurs on the appropriate position in R. Let this relation be denoted by $P \sqsubseteq R$, and $P \sqsubset R$ if $|P| < |R|$.

2D exact occurrence: P occurs at position (i, j) in T, $P \in A^{(m \times m')}$, $T \in A^{(n \times n')}$, if and only if $P[1..m; 1..m'] = T[i..i + m - 1; j..j + m' - 1]$.

A 2D factor of picture T is picture F such that $F \sqsubseteq T$. A 2D prefix of picture T is picture F_p, $F_p \sqsubseteq T$ and F_p occurs in T at position $(1, 1)$. A 2D suffix of picture T is picture F_s, $F_s \sqsubseteq T$, and F_s occurs in T at position $(n - m + 1, n' - m' + 1)$.

The following definitions introduce pushdown automata and related notions. A (nondeterministic) pushdown automaton (PDA) M, is a septuple $M = (Q, A, G, \delta, q_0, Z_0, F)$, where Q is a finite set of states, A is a finite input alphabet, G is a finite pushdown store alphabet, δ is a mapping $Q \times (A \cup \{\varepsilon\}) \times G \mapsto \mathcal{P}(Q \times G^*)$, $q_0 \in Q$ is an initial state, $Z_0 \in G$ is the initial pushdown store symbol, $F \subseteq Q$ is a set of final states. A pushdown store operation of PDA M, $M = (Q, A, G, \delta, q_0, Z_0, F)$, is a relation $(A \cup \{\varepsilon\}) \times G \mapsto G^*$. A pushdown store operation produces new contents on the top of the pushdown store by taking one input symbol or the empty string from the input and the current contents on the top of the pushdown store. The pushdown store grows to the right if written as string x, $x \in G^*$. A transition of PDA M is the relation $\vdash_M \subseteq (Q \times A^* \times G) \times (Q \times A^* \times G^*)$. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$. The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively.

A PDA is a deterministic PDA if it holds:

1. $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in A \cup \{\varepsilon\}$, $\gamma \in G$.
2. For all $\alpha \in G$, $q \in Q$, if $\delta(q, \varepsilon, \alpha) \neq \emptyset$, then $\delta(q, a, \alpha) = \emptyset$ for all $a \in A$.

A language L accepted by PDA M is a set of words over finite alphabet A. It is defined in two distinct ways: $L(M) = \{x;\ \delta(q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \gamma), x \in A^*, \gamma \in G^*, q \in F\}$ (*Acceptance by final state*), and $L_\varepsilon(M) = \{x;\ (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon), x \in A^*, q \in Q\}$ (*Acceptance by empty pushdown store*). If the PDA accepts the language by empty pushdown store then the set F of its final states is the empty set.

3 Representation of Multidimensional Arrays for Indexing

As noted e.g. in Giancarlo and Grossi [2], the first indexing data structure for arrays has been proposed by Gonnet [3]. He first introduced a notion of suffix trees for a 2D text (a PAT-tree) using a decomposition into a spiral string. Giancarlo [4] presented the *L-suffix tree* as a generalization of the suffix tree to square pictures and obtained a 2D suffix array from L-suffix trees. L-string equivalent notion has been described independently by Amir and Farach [5]. The time complexity of the construction is $\mathcal{O}(n^2 \log n)$, using $\mathcal{O}(n^2)$ space for a picture of $(n \times n)$. The same time complexity is reached also in Kim, Kim and Park [6]. Na, Giancarlo and Park [7] presented on-line version with the same construction time complexity, $\mathcal{O}(n^2 \log n)$, optimal for unbounded alphabets.

Kim, Kim and Park [8,6] presented an indexing structure for pictures (*I-suffix*), extensible to higher dimensions. They used a *Z-suffix* structure for indexing in three dimensional space. The only limitation is the pictures must be square and cubic, respectively, i.e. $(n \times n)$ and $(n \times n \times n)$, respectively. The presented $\mathcal{O}(n^2 \log n)$ time construction algorithm for direct construction of I-suffix arrays and $\mathcal{O}(n^3 \log n)$ time construction algorithm for Z-suffix arrays. This is the first algorithm for three-dimensional index data structure.

Giancarlo and Grossi [9,2], besides they gave an expected linear-time construction algorithm for square pictures, presented the general framework of 2D suffix tree families. E.g. [2, Def. 10.18] defines an index for a picture as a rooted tree whose arcs are labeled by *block characters*. A block character is in general some sub-picture of the given picture, its shape need not be rectangular. Similarly to tries used in string processing, these three conditions hold for such an index:

1. No two arcs leaving the same node are labeled by equal block characters.
2. For each sub-picture P there exists at least one node on the path from the root and concatenated labels on this path form this sub-picture P.
3. For any two nodes u and v, v being the ancestor of u, and $P(u), P(v)$ being the sub-pictures made of concatenated labels from the root to u, v, respectively, $P(v) \sqsubset P(u)$.

A suffix tree for pictures is then made of this index so that maximal paths of one-child nodes are compressed into single arcs.

Moreover, they proved the lower bound on the number of nodes of the suffix tree for 2D pictures: $\Omega(n^2 n')$, where $(n \times n')$ is the size of a picture, and $n \leq n'$. These suffix trees can be built in $\mathcal{O}(n^2 n' \log n')$ time and stored in optimal $\mathcal{O}(n^2 n')$ space [2, Theorem 10.22]. It means the construction of suffix tree for a square picture is easier than for an arbitrary sized picture. These results, used in [2], originated in Giancarlo [10].

The generalized index for structures of d dimensions is due to Giancarlo and Grossi [11]. They present *raw* and *heterogeneous* index that differ in work optimality of algorithms searching in them. Additionally, their method allows *dimensional wildcards*, that is, the 2D pattern may have less dimensions than is the number of dimensions of the indexed 2D text.

4 Tree-Based Decomposition of a Picture

Let us describe basic principles which our method is built upon. Our method uses vertex splitting to build a tree representing the picture. *Vertex splitting* is our informal name for a technique, where (some) nodes of a picture or a tree may appear multiple times in resulting data structure. Such a technique is not new, however. As an example, let us recall the construction of Kim, Kim and Park [8], recursively dividing picture into sub-pictures, starting at the origin and taking bottom and right sub-picture with the respect to the current vertex. We use the same principle.

Let us have a picture over alphabet A. First of all, we extend this picture with a special border symbol #, $\# \notin A$. Let us denote the alphabet of pictures extended this way by $A_{\#}$, $A_{\#} = A \cup \{\#\}$. This technique is commonly used in several other methods, namely as an input for two-dimensional on-line tessellation acceptors [12,1]. We use this special symbol to detect the end of a column and end of a row of a picture.

In Alg.1, a picture is extended only with minimum required additional symbols #. Since our method has no use for the top or left border made of symbol #, we avoid to augment the picture with unneeded borders.

Algorithm 1: Construction of an extended 2D picture
Input: Let P be a picture, $P \in A^{(n \times n')}$. Let $A_{\#} = A \cup \{\#\}$ be the ranked alphabet, where arity$(\#) = 0$, arity$(a) = 2$ for all symbols a, $a \in A_{\#}, a \neq \#$.
Output: Picture P' extended on its right and bottom border with symbol #, $P' \in A_{\#}^{((n+1)\times(n'+1))}$. (Element $[n+1, n'+1]$ is left undefined, it is never used.)
Method:
$P'[n+1, y] = \#$ for all y, where $1 \leq y \leq n'$,
$P'[x, n'+1] = \#$ for all x, where $1 \leq x \leq n$.

Next, Alg.2 for transformation of a picture into a tree is presented, its application to picture P will be denoted by tree(P).

Algorithm 2: Construction of a tree representation of a two-dimensional picture
Input: Picture P extended with symbol # (Alg.1), $P \in A_{\#}^{**}$, $P \neq \lambda$.
Output: Ordered labeled binary tree t, each node of t be labeled by a symbol s, $s \in A_{\#}$.
Description: Label, left and right children of node v are assigned as a triplet: (label, left child, right child). The algorithm starts at the origin of P. The application of the algorithm with P being its input will be denoted by tree(P).
Method:

$$\text{newnode}(x,y) = \begin{cases} (P[x,y], \text{newnode}(x,y+1), \text{newnode}(x+1,y)) & \text{, if } P[x,y] \neq' \#', \\ (P[x,y], nil, nil) & \text{, otherwise,} \end{cases}$$

$x = 1, y = 1,$
$t = (P[x,y], \text{newnode}(x,y+1), \text{newnode}(x+1,y))$.

Algorithm 2 is correct: it starts at the picture origin and recursively traverses the bottom and the right sub-pictures. Since the bottom row and the rightmost column are filled with symbols #, the recursion ends right at the leaves of the tree being constructed.

Lemma 1. *Let t be a tree constructed by Alg. 2. The maximal depth of any node in t,* $\mathrm{depth}(t)$*, is* $\mathrm{depth}(t) = n + n' - 1$.

Lemma 2. *The asymptotical time complexity of Alg. 2 for construction of the tree representation of a picture is* $\mathcal{O}(2^{\mathrm{depth}(t)})$.

In Janoušek [13], there are mentioned important properties of trees in the prefix notation.

Lemma 3 ([13], Theorem 4). *Given an ordered tree t and its prefix notation* $\mathrm{pref}(t)$*, all subtrees of t in the prefix notation are substrings of* $\mathrm{pref}(t)$.

Not every substring of a tree in the prefix notation is a prefix notation of its subtree. This can be seen from the fact that for a given tree with n nodes there can be $\mathcal{O}(n^2)$ distinct substrings, but there are just n subtrees. Each node of the tree is the root of just one subtree. Only those substrings which themselves are trees in the prefix notation are those which are the subtrees in the prefix notation. This property is formalised in [13, Theorem 6].

Let us define the arity checksum of a string over a ranked alphabet first: Let $w = a_1 a_2 \cdots a_m$, $m \geq 1$, be a string over ranked alphabet A. The arity checksum $\mathrm{ac}(w)$ of string w is then $\mathrm{ac}(w) = \mathrm{arity}(a_1) + \mathrm{arity}(a_2) + \cdots + \mathrm{arity}(a_m) - m + 1 = \sum_{i=1}^{m} \mathrm{arity}(a_i) - m + 1$.

Lemma 4 ([13], Theorem 6). *Let* $\mathrm{pref}(t)$ *be the prefix notation of tree t, w be a substring of* $\mathrm{pref}(t)$*. Then, w is the prefix notation of a subtree of t if and only if* $\mathrm{ac}(w) = 0$*, and* $\mathrm{ac}(w_1) \geq 1$ *for each w_1 where $w = w_1 x$, $x \neq \varepsilon$.*

In tree pattern and subtree pushdown automata the arity checksum is computed by pushdown operations. The content of the pushdown store represents the corresponding arity checksum. The empty pushdown store means that the corresponding arity checksum is equal to zero.

Let us extend these results to our representation of pictures in the prefix notation. For the following three theorems: Let T be a picture and T' be T extended by Alg. 1. Let t be a tree constructed by Alg. 2, such that $t = \mathrm{tree}(T')$. Let P be a sub-picture of T, s be a tree made over extended P.

Theorem 5. *For every $P \sqsubseteq T$, such that P is a 2D prefix of T, it holds that* $\mathrm{pref}(s)$ *is a subsequence of* $\mathrm{pref}(t)$ *in general and its first symbol is the first symbol of* $\mathrm{pref}(t)$.

Theorem 6. *For every $P \sqsubseteq T$, such that P is a 2D suffix of T, it holds that* $\mathrm{pref}(s)$ *is a substring of* $\mathrm{pref}(t)$.

Theorem 7. *For every $P \sqsubseteq T$, such that P is a 2D factor of T, it holds that* $\mathrm{pref}(s)$ *is a subsequence of* $\mathrm{pref}(t)$.

5 Two-Dimensional Pattern Matching in Pictures in the Tree Representation

We have seen the representation of a picture based on the prefix notation of a tree allows to use tree matching algorithms to locate two-dimensional pattern transformed into the same representation. There are numerous methods for various kinds of matching in trees, they are described in Comon *et al.* [14] and Cleophas [15].

Another possibility is to use the prefix notation of the tree transformed pictures. This variant allows to use some of the methods developed in [13]. Specifically for trees in prefix notation, the tree pattern pushdown automata (from [13]) are analogous to factor automata used in string matching, they make an index from the tree (in our case a 2D text is transformed into a tree) in the prefix notation and then the pattern in the prefix notation is searched in it. The tree pattern pushdown automaton finds the rightmost leaves of all occurrences of a tree pattern in the subject tree. The searching for a 2D prefix, 2D suffix, and a 2D factor in our tree representation is equivalent to a treetop matching, a subtree matching, and a tree template matching, respectively.

The most significant advantage over other tree algorithms is that for given input tree pattern of size m, the tree pattern PDA performs its searching in time linear in m, independently of the size n of the subject tree. This is faster than the time that could be theoretically achieved by any standard tree automaton, because the standard deterministic tree automaton runs on the subject tree. As a consequence, the searching performed by the tree automaton can be linear in n at best.

The tree pattern PDA is by construction nondeterministic input-driven PDA. Any input-driven PDA can be determinised [18]. However, there exists a method constructing deterministic tree pattern pushdown automata directly from tree patterns [16].

Using some tree algorithm or tree matching pushdown automaton, in either case the size of the tree representation of a picture will be prohibitive. In the former case, it will influence the time of matching, in the latter, the pushdown automaton becomes large. The redundancy in the PDA is caused by repeated subtrees in the tree representation.

In the following, we will show a pushdown automata based model for picture indexing, addressing some of these problems.

6 Pushdown Automata for Picture Indexing

In this section, a pushdown automata based picture index will be presented. It reuses its own parts in a manner resembling the lateral gene transfer links of Newick notation [17]. This means some part of a genome of some species is reused in the genome of some other species, non-related in terms of the standard line of ancestry.

The high number of redundant subtrees, induced by the picture-to-tree transformation, can be eliminated by redundant subtree removal and by addition of edges leading to roots of appropriate subtrees. These additional transitions make a directed acyclic graph (DAG) from the tree, however.

Another interesting property is that the pushdown automaton construction does not need the tree representation of a picture constructed and available. The pushdown automaton can be constructed from the picture directly. Since the tree representation of a 2D pattern can also be generated on the fly, the only significant extra data structure in memory will be the indexing pushdown automaton. On the other hand, we pay for the reusability of pushdown automata components by increase in the number of pushdown store symbols. This converts the category of the PDA from input-driven to nondeterministic PDA. It is not known yet if it is possible to transform these nondeterministic pushdown automata to deterministic pushdown automata.

6.1 Two-Dimensional Pattern Matching Using Pushdown Automata

In this section, a construction of picture indexing automata will be discussed. These automata allow to accept all 2D prefixes, 2D factors and 2D suffixes of a given 2D text in the prefix notation. The deterministic indexing pushdown automaton accepts the prefix notation of a tree made of 2D prefix, 2D factor, or 2D suffix, in time linear to the number of nodes of the tree. The type of 2D pattern accepted is given by construction of the pushdown automaton.

The construction of a pushdown automaton for 2D text indexing is based on linearisation of a picture with adjacency relation among array elements preserved.

First algorithm in this section constructs a base for three variants of pushdown automata presented later.

Algorithm 3: Pushdown automaton accepting the picture in prefix notation

Input: Picture T extended with symbol $\#$, $T \in A_{\#}^{((n+1)\times(n'+1))}$.

Output: Automaton M, $M = (Q, A_{\#}, G, \delta, q_{1,1}, S, \emptyset)$, accepting language $L_\varepsilon(M)$, $L_\varepsilon(M) = \{\text{pref}(\text{tree}(T))\}$.

Method:

$G = \{S\}$, $Q = \emptyset$, $\delta(q, a, z) = \emptyset$ for all $a \in A_{\#} \cup \{\varepsilon\}, z \in G$,

$G = G \cup \{S_{x+1,y}\}$ for all x, y, where $1 \le x < n$, $1 \le y < n'$,

$Q = Q \cup \{q_{x,y}\}$ for all x, y, where $1 \le x \le n+1$, $1 \le y \le n'+1$ and $y < n' \vee x < n$,

$\delta(q_{x,y}, T[x,y], S) = \delta(q_{x,y}, T[x,y], S) \cup \{(q_{x,y+1}, S_{x+1,y}S)\}$ for all x, y, where $1 \le x < n$, $1 \le y < n'$,

$\delta(q_{n,y}, \#, S) = \delta(q_{n,y}, \#, S) \cup \{(q_{n+1,y}, \varepsilon)\}$ for all y, where $1 \le y < n'$,

$\delta(q_{x,n'}, \#, S) = \delta(q_{x,n'}, \#, S) \cup \{(q_{x,n'+1}, \varepsilon)\}$ for all x, where $1 \le x \le n$,

$\delta(q_{x,n'+1}, \varepsilon, S_{x+1,y}) = \delta(q_{x,n'+1}, \varepsilon, S_{x+1,y}) \cup \{(q_{x+1,y}, S)\}$ for all x, y, where $1 \le x < n$, $1 \le y < n'$,

$\delta(q_{n+1,y}, \varepsilon, S_{x,y-1}) = \delta(q_{n+1,y}, \varepsilon, S_{x,y-1}) \cup \{(q_{x,y-1}, S)\}$ for all x, y, where $2 \le x \le n$, $2 \le y < n'$.

The following algorithms add transitions allowing the automaton M to accept the prefix notation of trees representing 2D suffixes and 2D prefixes.

Algorithm 4: Suffix transitions

Input: Picture T extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$. Pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, constructed by Alg.3.
Output: Modified pushdown automaton M.
Method:
$\delta(q_{1,1}, T[x,y], S) = \delta(q_{1,1}, T[x,y], S) \cup \{(q_{x,y+1}, S_{x+1,y}S)\}$ for all x, y, where $1 \leq x < n$, $1 \leq y < n'$ and $x > 1 \vee y > 1$.

Algorithm 5: Prefix end transitions

Input: Picture T extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$. Pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, constructed by Alg.3.
Output: Modified pushdown automaton M.
Method:
$\delta(q_{x,y}, \#, S) = \delta(q_{x,y}, \#, S) \cup \{(q_{n+1,y}, \varepsilon)\}$ for all x, y, where $1 \leq x < n$, $1 \leq y < n'$ and $x \neq 1 \vee y \neq 1$.

Algorithm 6: Construction of pushdown automaton accepting all 2D factors

Input: Picture T extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$.
Output: Automaton M, $M = (Q, A_{\#}, G, \delta, q_{1,1}, S, \emptyset)$, accepting language $L_\varepsilon(M)$, $L_\varepsilon(M) = \{w; w = \text{pref}(\text{tree}(\text{2D factor of } T))\}$.
Method:
Create pushdown automaton M using Alg.3.
Add suffix transitions to its δ using Alg.4.
Add prefix transitions to its δ using Alg.5.

Algorithm 7: Construction of pushdown automaton accepting all 2D prefixes

Input: Picture T extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$.
Output: Automaton M, $M = (Q, A_{\#}, G, \delta, q_{1,1}, S, \emptyset)$, accepting language $L_\varepsilon(M)$, $L_\varepsilon(M) = \{w; w = \text{pref}(\text{tree}(\text{2D prefix of } T))\}$.
Method:
Create pushdown automaton M using Alg.3.
Add prefix transitions to its δ using Alg.5.

Construction of Pushdown Automata for 2D Text Indexing. The correctness of the 2D factor pushdown automaton, constructed by Alg.6 and accepting 2D factors of a picture in prefix notation, is established by the following theorem.

Theorem 8. *Let T be a picture extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$, pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, be constructed by Alg.6 over T. Pushdown automaton M accepts prefix notation of any 2D factor of T by the empty pushdown store. That is, M accepts the prefix notation of any $T[i..k, j..l]$, $1 \leq i < k \leq n$, $1 \leq j < l \leq n'$.*

The correctness of the 2D prefix pushdown automaton, constructed by Alg.7 and accepting 2D prefixes of a picture in prefix notation, is established by the following theorem.

Algorithm 8: Construction of pushdown automaton accepting all 2D suffixes
Input: Picture T extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$.
Output: Automaton M, $M = (Q, A_{\#}, G, \delta, q_{1,1}, S, \emptyset)$, accepting language $L_\varepsilon(M)$, $L_\varepsilon(M) = \{w; w = \text{pref}(\text{tree}(\text{2D suffix of } T))\}$.
Method:
Create pushdown automaton M using Alg.3.
Add suffix transitions to its δ using Alg.4.

Theorem 9. *Let T be a picture extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$, pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, be constructed by Alg.7 over T. Pushdown automaton M accepts prefix notation of any 2D prefix of T by the empty pushdown store. That is, M accepts the prefix notation of any $T[1..k, 1..l]$, $1 \leq k \leq n$, $1 \leq l \leq n'$.*

The correctness of the 2D suffix pushdown automaton, constructed by Alg.8 and accepting 2D suffixes of a picture in the prefix notation, is established by the following theorem.

Theorem 10. *Let T be a picture extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$, pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, be constructed by Alg.8 over T. Pushdown automaton M accepts the prefix notation of any 2D suffix of T by the empty pushdown store. That is, M accepts the prefix notation of any $T[i..n, j..n']$, $1 \leq i \leq n$, $1 \leq j \leq n'$.*

Lemma 11. *Let T be a picture extended with symbol #, $T \in A_{\#}^{((n+1)\times(n'+1))}$, 2D prefix pushdown automaton M, $M = (Q, A, G, \delta, q_{1,1}, S, \emptyset)$, be constructed by Alg.7 over T. Pushdown automaton M is the deterministic PDA.*

7 Conclusion

In this paper, a method of 2D exact pattern matching using pushdown automata has been presented. We have shown how a picture can be transformed into a tree, where the context of each element of the picture is preserved. The tree can be linearised into its prefix notation and a pattern matching can be performed over this representation.

All presented pushdown automata are indexing the 2D text and allow to search for the 2D pattern in time proportional to the size of the pattern itself, independently of the size of the text. 2D factor and suffix automata are nondeterministic. However, 2D prefix automaton is deterministic and optimal in size, since its size is proportional to the size of a 2D text. The lower bound for storing nodes of the suffix tree for pictures is $\Omega(n|T|)$, the best construction algorithm runs in $\mathcal{O}(|T| \log n)$ for square pictures. Our algorithm for construction of 2D prefix automaton runs in $\mathcal{O}(|T|)$ time for a bounded alphabet and the automaton has $\mathcal{O}(|T|)$ states.

Acknowledgements. The authors wish to acknowledge the helpful comments of Jan Lahoda and anonymous referees.

References

1. Giammarresi, D., Restivo, A.: Two-dimensional languages. In: Handbook of Formal Languages, vol. III, pp. 216–267. Springer, Heidelberg (1997)
2. Giancarlo, R., Grossi, R.: Suffix tree data structures for matrices. In: Apostolico, A., Galil, Z. (eds.) Pattern Matching Algorithms, pp. 293–340. Oxford University Press, Oxford (1997)
3. Gonnet, G.H.: Efficient searching of text and pictures. Report OED-88-02, University of Waterloo (1988)
4. Giancarlo, R.: A generalization of the suffix tree to square matrices, with applications. SIAM J. Comput. 24(3), 520–562 (1995)
5. Amir, A., Farach, M.: Two-dimensional dictionary matching. Inf. Process. Lett. 44(5), 233–239 (1992)
6. Kim, D.K., Kim, Y.A., Park, K.: Generalizations of suffix arrays to multi-dimensional matrices. Theor. Comput. Sci. 302(1-3), 401–416 (2003)
7. Na, J.C., Giancarlo, R., Park, K.: On-line construction of two-dimensional suffix tree in $\mathcal{O}(n^2 \log n)$ time. Algorithmica 48, 173–186 (2007)
8. Kim, D.K., Kim, Y.A., Park, K.: Constructing suffix arrays for multi-dimensional matrices. In: Farach-Colton, M. (ed.) CPM 1998. LNCS, vol. 1448, pp. 126–139. Springer, Heidelberg (1998)
9. Giancarlo, R., Grossi, R.: On the construction of classes of suffix trees for square matrices: Algorithms and applications. Inf. Comput. 130(2), 151–182 (1996)
10. Giancarlo, R.: An index data structure for matrices, with applications to fast two-dimensional pattern matching. In: Dehne, F., et al. (eds.) WADS 1993. LNCS, vol. 709, pp. 337–348. Springer, Heidelberg (1993)
11. Giancarlo, R., Grossi, R.: Multi-dimensional pattern matching with dimensional wildcards: Data structures and optimal on-line search algorithms. J. Algorithms 24(2), 223–265 (1997)
12. Inoue, K., Nakamura, A.: Some properties of two-dimensional on-line tessellation acceptors. Inf. Sci. 13(2), 95–121 (1977)
13. Janoušek, J.: String suffix automata and subtree pushdown automata. In: Holub, J., Žd'árek, J. (eds.) Proc. PSC 2009, CTU in Prague, Czech Republic, pp. 160–172 (2009)
14. Comon, H., et al.: Tree automata techniques and applications (2007), http://www.grappa.univ-lille3.fr/tata (release October 12, 2007)
15. Cleophas, L.: Tree Algorithms. Two Taxonomies and a Toolkit. PhD thesis, Technische Universiteit Eindhoven, Eindhoven (2008)
16. Flouri, T., Janoušek, J., Melichar, B.: Tree pattern matching by deterministic pushdown automata. In: Ganzha, M., Paprzycki, M. (eds.) Proc. IMCSIT, vol. 4, pp. 659–666. IEEE Computer Society Press, Los Alamitos (2009)
17. Olsen, G.: "Newick's 8:45" tree format standard (August 1990), http://evolution.genetics.washington.edu/phylip/newick_doc.html
18. Wagner, K., Wechsung, G.: Computational Complexity. Springer, Heidelberg (2001)
19. Žd'árek, J.: Two-dimensional Pattern Matching Using Automata Approach. PhD thesis, Czech Technical University in Prague (2010), http://www.stringology.org/papers/Zdarek-PhD_thesis-2010.pdf

Regular Expressions at Their Best: A Case for Rational Design

Vincent Le Maout

Exalead SA, 10 place de la Madeleine, 75008 Paris, France
vincent.le-maout@exalead.com
http://www.exalead.com

Abstract. Regular expressions are often an integral part of program customization and many algorithms have been proposed for transforming them into suitable data structures. These algorithms can be divided into two main classes: backtracking or automaton-based algorithms. Surprisingly, the latter class draws less attention than the former, even though automaton-based algorithms represent the oldest and by far the fastest solutions when carefully designed. Only two open-source automaton-based implementations stand out: PCRE and the recent RE2 from Google. We have developed, and present here, a competitive automaton-based regular expression engine on top of the LGPL C++ Automata Standard Template Library (ASTL), whose efficiency and scalability remain unmatched and which distinguishes itself through a unique and rigorous STL-like design.

Keywords: regular expression, automata, C++, ASTL, generic programming, efficiency, scalability.

1 Introduction

There are many real-life areas in which data grows at the same rate as or faster than computing power. Textual data processing of web pages is one such example. In these areas, implementation efficiency of basic algorithms still remains a major concern. One basic processing step in many text processing tasks involves using regular expressions to recognize a sub language. For example, in a web search engine, some type of implementation of regular expressions or transducers [2] are used at document indexing time or at query processing time to perform a variety of natural language processing functions: tokenization, named-entity detection, acronym, URL, email address detection, sentence detection, input detection for rule base systems, etc. Regular expressions are used extensively in search engines: search engine developers use regular expressions on a daily basis; administrators might tune the engine behavior by writing regular expression based rules that exclude uninteresting parts of documents during indexing; and some search engines, such as Exalead's, even allow users to perform search queries involving regular expressions. As search continues to evolve with more added functionality and greater breadth, with constant expectations of rapid

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 310–320, 2011.

system response time, there is a sustained need to improve the efficiency and scalable treatment of more and more complex regular expressions.

Since regular expressions are used in so many response-time dependent tasks, processing time is an important constraint on the complexity of features that can be implemented with them. After unsuccessful trials during our search for a versatile, efficient, scalable implementation supporting all well-known features, we had to conclude that none of the available libraries met our requirements. We tested Boost::regex [17], Boost::xpressive [18], PCRE[1] [19], and Microsoft's GRETA [20]. Their implementations are all based on backtracking algorithms which seem to constitute the main problem with their performance. We decided to explore whether the neglected classical approach to regular expression implementation, based on seminal work by Thompson [9], might offer a way to overcome the difficulties in a simple manner.

2 Backtracking vs. Automaton-Based Implementations

Algorithms for implementing a regular expression matcher fall into two main categories, differing the treatment of non-deterministic automata (NFA). One technique for NFA performs depth-first iteration, trying every possible path from an initial state to a final state. This solution stores positions in a stack when a fork is reached in the automaton and *backtracks* to the latest position on the stack when a path does not lead to an accepting state. The other technique for NFA performs breadth-first iteration, maintaining a list of positions initialized with the set of initial states and finding a path to a set that contains at least one accepting state. This solution maintains sets of states in some appropriate data structure; these sets are then mapped to unique states in a deterministic automaton (DFA) whose transition accesses are highly optimized (the subset construction, [1], pp 118-120). The advantages of the latter DFA-based implementations, as opposed to the former backtracking option, are so numerous and drawbacks so few that one wonders why so few open-source libraries for this technique are available. Consider:

- DFA algorithms and data structures involved have been studied and theoretically explored since the 60's ([13], [9], [2], [1]).
- DFAs can find all matches in one pass, which makes them adapted to stream-based data processing.
- Most pathological cases of backtracking are avoided [10].
- They provide more freedom of use via incremental match and tunable matching policies.
- Greater efficiency and scalability with respect to the expression size are achieved.

Backtracking solutions mainly offer the advantage of ease of implementation, sparing the developer theoretical considerations (and the need for clean design):

[1] PCRE has two running modes, backtracking or automaton-based but the latter does not provide capturing and exhibits poor performances.

- Some functionalities are more easily implemented via backtracking: submatching, laziness (greediness) of quantifiers and backreferences.
- They avoid constructing intermediate NFA with epsilon transitions (these allow the automaton to change state without consuming an input character, matching the empty word ϵ).
- DFA-based implementations also require lazy determinization, especially when dealing with multi-byte Unicode, which may be tricky to implement without sound design principles.

Our non-backtracking implementation, described below, uses the simplest algorithms known, avoiding stumbling blocks that are usually used as spurious excuses for using backtracking architectures.

2.1 Avoiding the Construction of NFA

The construction of a deterministic automaton from a regular pattern has two main stages: first, building a NFA with epsilon transitions, then, building from this NFA, a DFA in which each state corresponds to a subset of the NFA states. Actually, we can avoid the NFA construction with the algorithm described in [1], pp 135-141, leading to a simpler implementation: during the pattern parsing, each node of the produced tree is decorated with pointers that link to the next positions in the pattern when an input character is matched. Sets of these pointers constitute the DFA transitions, and eliminate the need for an extra data structure and its associated algorithms (the so-called Thompson's construction requires a NFA with epsilon transitions and the epsilon-closure, [1], pp 118-120).

2.2 Lazy Determinization

The DFA acts as a cache of transitions, eliminating the need for repeated and inefficient accesses to the non-deterministic data structure: transitions are added when needed, and computed once and for all, dramatically improving speed when most of them are constructed. This technique can also avoid huge memory consumption: for example, a match-all dot will only expand to a few dozens transitions when scanning an English text instead of tens of thousands when the DFA is entirely constructed to handle Unicode. (This lazy aspect is briefly addressed in ([1], p 128) but, then, most of the texts were encoded in ASCII and the issue was not so crucial.) This technique also improves response time since computation of construction is spread over the matching time, reducing initialization time to nearly zero.

It should be noted that GNU egrep implements a lazy construction algorithm and shows impressive performances on ASCII text. It is however quite inefficient in UTF-8 mode: simple tests have shown that it is up to 700 times slower, even if the input actually contains only ASCII characters.

2.3 Pathological Cases Avoided

Pathological cases arise when depth-first iteration in a NFA has to try so many paths that the time required is exponential, which may occur with very simple

expressions [10]. With a lazily constructed DFA, since matching is done in one pass, in linear time, for a pathological case to occur a combination of both a pathological expression *and* pathological text must occur, which is very rare in practice. In this case, the size of the DFA can grow unreasonably, up to 2^n transitions where n is the number of transitions in the NFA. Simply discarding parts of the DFA can limit its size and because at worst one transition is added for each input character, the processing time remains linear in the size of the text. Removing transitions slows things down since they may need to be computed again later but as this processing does not create a practically infinite loop, the DFA size is strictly bounded and the result of the algorithm is well-defined.

In addition, a DFA-based implementation does not use a stack, so one avoids the risk of backtrack stack overflow, which can lead to undefined behavior in Perl (depending on the version, either a segmentation fault or an unpredictable result when the algorithm aborts) and which may represent a risk if memory consumption is not bounded.

2.4 More Freedom of Use

Commonly used backtracking algorithms impose some limitations that are easily overcome with a DFA-based processing:

- Matching a pattern in a data stream requires a one-pass algorithm which is not possible with backtracking.
- "Interrupted matching", i.e. the need for processing to start on one buffer and to continue on another one cannot be easily implemented. This case arises when dealing with huge texts that do not fit entirely in RAM or when matching is performed on a stream of tokens.
- DFA design allows easy matching against trees or automata since the algorithm may be incrementally applied (see Sect. 3.1).
- It is easy to let the user choose a matching policy (shortest/longest matches or all matches, overlapping or not) which is not possible with backtracking[2].

2.5 Submatching

Submatches are more easily implemented in backtracking algorithms and constitute the tricky part in an automaton-based processing, especially during determinization. This may explain why so many developers willing to provide submatching functionality choose backtracking and why the DFA mode of PCRE does not support it. Submatching with an automaton-based data structure was not seriously addressed, either theoretically and practically, until 2000 [15,14].

2.6 Efficiency and Scalability

Efficiency is clearly the most important criterium for judging of the interest of a regular-expression engine, and when it comes to speed, backtracking algorithms are no match for DFA-based ones (see Sect. 4 for a speed comparison). A DFA-based implementation is insensitive to expression size because its deterministic

[2] Backtracking has, however, the notion of greediness for quantifiers.

data structure provides constant-time access to transitions, thus making their number irrelevant to the computation time (see Sect. 4 about run time with respect to expression complexity).

3 Design and Implementation

The Automaton Standard Template Library (ASTL) [6], [8], [7] is the basis of our code because (i) it targets efficiency and reusability and (ii) the concepts it provides are well-defined and well-tested. Several automaton data structures are included and lazy construction is automatically provided. Recently, Google released RE2 [21] which tackles the problem with a virtual machine executing byte-code ([11], [12]), much like Thompson's original algorithms. Our implementation differs in that it does not involve code generation and execution by a machine (virtual or not) and may be considered as an abstraction of it. However, both of these libraries end up building a DFA lazily and consequently exhibit the same behavior and characteristics.

3.1 ASTL, Cursors and Incrementality

ASTL's design eases reusability and adaptability. Inspired by the three-layer model of generic programming [3] (algorithm, iterator, container), and the C++ Standard Template Library (STL, [5], [4]), it defines an abstract accessor, called a *cursor*, to automata containers. ASTL algorithms communicate only with the cursors, which uncouples processing from the data structures it operates on. By organizing algorithms into layers, with each layer responsible for a part of the algorithm performed on-the-fly, and by stacking cursors either through inheritance or aggregation, a generic architecture is achieved, providing powerful lazy construction automatically. Fig. 1 shows the architecture of the engine.

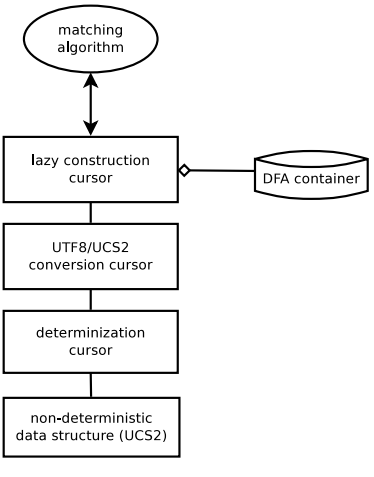


Fig. 1. ASTL-based architecture of regular expression matcher

A cursor is a pointer to a state of a DFA and can move along defined transitions in a very simple manner. Here is the abstract interface:

`bool forward(char c)`	try to move along the transition labelled with `c`
`bool final()`	tell if current state is an accepting state
`int state()`	retrieve the current state identifier
`void operator=(int s)`	set the cursor on the state `s`

Following is a simple example of use: the `match` function returns `true` when an input string matches a pattern.

```
bool match(cursor c, const char *first, const char *last) {
  while (first != last && c.forward(*first)) {
    if (c.final()) return true;
    ++first;
  }
  return false;
}
```

Note that this algorithm makes no assumptions on how the cursor `c` is implemented; there are no references to any kind of automaton or regular expressions.

Another real-world example is given by Exalead's query language which allows query matching by specifying regular expression patterns enclosed in slashes as well as mere character strings. In the following query:

```
/[ND]FA/ NEAR /implementations?/
```

`[ND]FA` and `implementations?` are expanded by a matching between a cursor and the trie of the index dictionary. A handful of code lines added to the depth-first traversal of the trie does the trick: the cursor simply signals the algorithm when the last call to `forward` failed and that is must pop the stack because the current branch is not in the language defined by the pattern.
Here is a sketch of the algorithm:

```
void search(node n, cursor c, vector<string> &results) {
  int s = c.state();                        // save cursor position
  // for each son of node n:
  for(node::iterator son = n.begin(); son != n.end(); ++son) {
    if (c.forward(son->letter())) {         // try move on the DFA
      results.back().push(son->letter());   // push current letter
      if (c.final())                        // accepting state?
        results.push_back(results.back()); // store matching word
      search(*son, c, results);             // search deeper
      results.back().pop();                 // pop current letter
    }
    c = s;                                  // set to saved position
  }
}
```

This incremental aspect constitutes an advantage of the ASTL API over RE2.

3.2 Encapsulation, Lazy Determinization, Submatches, Thread-Safety

As mentioned in Sect. 2.1, no NFA is built in the ASTL design, and this is hidden from rest of the code by a cursor interface. Should the need for a real NFA implementation arise, switching to a classical non-deterministic automaton construction with epsilon transitions would have no impact on other components.

Lazy determinization avoids the explosion of the data structure size when handling Unicode and when dealing with huge patterns. Whatever the encoding of input, a DFA works on bytes (no semantics at this level) which permits the use of a matrix structure with an optimal constant-time access to transitions, irrespective of their number: an array of fixed size is allocated per state, containing 256 state identifiers with the undefined transitions set to null, resulting in a run time proportional solely to the input data byte count. This makes matching slower over UTF16 or UCS2-encoded text and on several-byte UTF-8 characters but still faster than using an automaton container that would store up to thousands transitions per state in a hash table or a logarithmic-time access structure like a balanced tree. To handle different encodings, a cursor layer converts the input to the internal UCS2 representation and the lazy construction is provided by the ASTL framework under the form of a cursor with only 20 lines of code.

Capturing submatches requires maintaining a matrix of positions in the input text, with two lines per sub-expression (one for the left parenthesis, one for the right) and one column per position where values are added when crossing the corresponding parenthesis in the pattern or removed by disambiguation along the way [14]. The decision to add and/or remove values in this matrix is encoded in the DFA transitions, which makes it look more like a transducer than a pure automaton, though there is no notion of output language here.

The non-deterministic layer is a read-only structure, so it can be shared between threads without danger. The deterministic automaton however is thread-specific because it is dynamically constructed and locking the data structure for each input character constitutes an unacceptable overhead. This ASTL design avoids the use of any synchronization mechanics, which enhances portability since there is no standard mutexes or locks in C++.

4 Performance

The following tests compare libraries performance for speed and scalability. The results of PCRE in DFA mode are not reported because its performances were disapointing, well below the average, so **pcre** in the tests denotes its use in backtracking mode. Full details are available in [16].

Speed was measured for a variety of use-cases: matching small and big patterns on short strings or big texts. The times are reported relative to the fastest one,

which gets a **1**, to show results independent from the hardware. Here is an extract from the tests for short match:

astl	re2	regex	xpressive	greta	pcre	text	pattern
2.40	1.91	2.54	1.35	1	1.10	100- this is...	`^([0-9]+)(\-| |$)(.*)$`
1	1.61	3.03	1.92	1.96	1.53	1234-5678-...	`([0-9]{4}[- ]){3}[0-9]{3,4}`
1	4.37	8	3	2.12	3.5	123	`^[-+]?[0-9]*\.?[0-9]*$`
1	2.46	5.4	1.66	1.33	2	+3.14159	`^[-+]?[0-9]*\.?[0-9]*$`

Patterns starting with literal strings are of particular interest since there are many simple ways to greatly enhance the speed in these cases: our implementation is optimized with the trivial Horspool variant of Boyer-Moore algorithm on UCS2 ([22,23,24]) and the standard C function `memchr` over UTF-8, thus achieving up to several gigabytes per seconds. Input (complete works of Mark Twain):

astl	re2	regex	xpressive	greta	pcre	pattern
1	1.5	8.5	7	7	6	`Twain`
1	1	8	7	7	5.5	`Huck[[:alpha:]]+`
1	6.71	3.28	3.57	15.42	4.28	`Tom|Sawyer|Huckleberry|Finn`

We then calculated the average score over all tests for each library. An average value of **1** means that an implementation is the fastest in all tests:

library	overall average
astl	**1.34041**
re2	1.89456
boost::xpressive	2.80061
greta	3.65633
pcre	4.00644
boost::regex	4.16841

The scalability test evaluates the behavior of the compared libraries when the complexity of the expression increases. Starting from a simple expression, at each step, a new expression is concatenated as an alternative and the time to find all matches in roughly 200Mb of English text is measured. Figure 2 shows the behavior of backtracking implementations. Figure 3 shows that for DFA-based implementations, the size of the pattern is irrelevant, provided that the transition cache is given enough time to warm-up[3]. This is due to the constant-time access to the transitions. The scale has been preserved between those two graphs to highlight the huge difference in the run time behavior.

[3] Most of the time, only a few hundred bytes are needed to reach a stable size for the DFA.

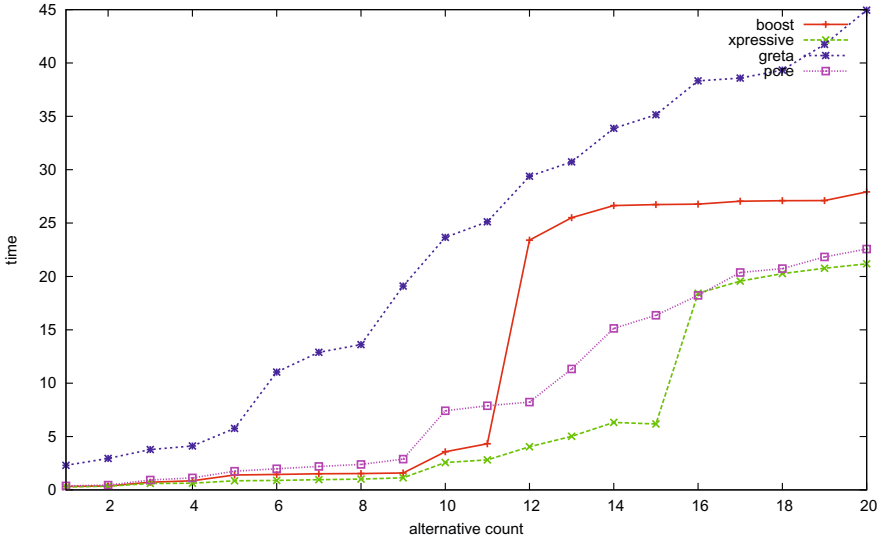


Fig. 2. Matching time for backtracking implementations with respect to expression complexity

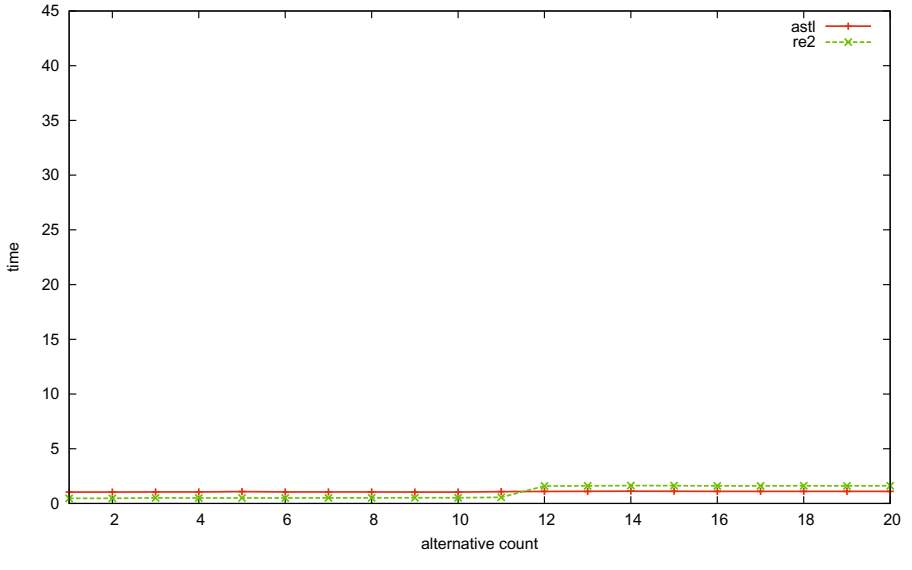


Fig. 3. Matching time for DFA-based implementations with respect to expression complexity

5 Conclusion

DFA-based implementations for regular expression matching demonstrate a clear superiority over backtracking implementations. The use of a lazy DFA construction, whose data structure has optimal transition access, bounded size, and that works on bytes in linear time, whatever the pattern, allows us to cope with present-day constraints on ever-growing sizes of data and expressions, Unicode support, variety of text encodings, providing predictability and robustness of the processing. Their recent availability in open-source code (a recency which is quite surprising given that the algorithms involved have been well-known for years) should move developers to adopt them in their first implementation of regular expression processing, or prompt them to switch from backtrack-based implementations. This work shows that, through clean software design rooted in solid theroretical and technical considerations, most of the crippling obstacles may be overcome with simple solutions, leading to optimal and sustainable code, viable in real-world and highly-constrained environments.

Acknowledgments

This work was partly realized as part of the Quaero Programme, funded by OSEO, French State agency for innovation.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers - Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Hopcroft, J.E., Ullman, J.D.: Introduction to automata, languages and computation. Addison-Wesley, Reading (1979)
3. Musser, D.R., Stepanov, A.: Generic Programming. In: Gianni, P. (ed.) ISSAC 1988. LNCS, vol. 358. Springer, Heidelberg (1989)
4. Standard Template Library Programmer's Guide, Silicon Graphics Computer Systems (1999), `http://www.sgi.com/Technology/STL`
5. Stepanov, A., Lee, M.: The Standard Template Library. HP Laboratories Technical Report 95-11(R.1) (1995)
6. Le Maout, V.: Tools to Implement Automata, a first step: ASTL. In: Wood, D., Yu, S. (eds.) WIA 1997. LNCS, vol. 1436, pp. 104–108. Springer, Heidelberg (1998)
7. Le Maout, V.: PhD Thesis: Expérience de programmation générique sur des structures non-séquentielles: les automates, Université de Marne-La-Vallée (2003)
8. Le Maout, V.: Cursors. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 195–207. Springer, Heidelberg (2001)
9. Thompson, K.: Regular expression search algorithm. CACM 11(6) (1968)
10. Cox, R.: Regular Expression Matching Can Be Simple And Fast (2007), `http://swtch.com/~rsc/regexp/regexp1.html`
11. Cox, R.: Regular Expression Matching: the Virtual Machine Approach (2009), `http://swtch.com/~rsc/regexp/regexp2.html`
12. Cox, R.: Regular Expression Matching in the Wild (2010), `http://swtch.com/~rsc/regexp/regexp3.html`

13. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IRE Transactions on Electronic Computers EC-9(1), 39–47 (1960)
14. Laurikari, V.: Efficient Submatch Addressing for Regular Expressions (2001)
15. Laurikari, V.: NFAs with Tagged Transitions, their Conversion to Deterministic Automata and Application to Regular Expressions (2000)
16. Le Maout, V.: Regular Expression Performance Comparison (2010), http://astl.sourceforge.net/bench.7.html
17. Maddock, J.: Boost Regex (2007), http://www.boost.org/doc/libs/1_42_0/libs/regex/doc/html/index.html
18. Niebler, E.: Boost Xpressive (2007), http://boost-sandbox.sourceforge.net/libs/xpressive/doc/html/index.html
19. PCRE. Univ. Cambridge (2009), http://sourceforge.net/projects/pcre/
20. Niebler, E.: GRETA. Microsoft (2003), http://research.microsoft.com/en-us/downloads/BD99F343-4FF4-4041-8293-34C054EFE749/default.aspx
21. Cox, R.: RE2, Google (2010), http://code.google.com/p/re2/
22. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. CACM 20(10), 762–772 (1977)
23. Horspool, R.N.: Practical fast searching in strings. Software - Practice & Experience 10, 501–506 (1980)
24. Baeza-Yates, R.A., Régnier, M.: Average running time of the Boyer-Moore-Horspool algorithm. Theoretical Computer Science 92(1), 19–31 (1992)

Simulations of Weighted Tree Automata

Zoltán Ésik[1,⋆] and Andreas Maletti[2,⋆⋆]

[1] University of Szeged, Department of Computer Science
Árpád tér 2, 6720 Szeged, Hungary
ze@inf.u-szeged.hu
[2] Universitat Rovira i Virgili, Departament de Filologies Romàniques
Avinguda de Catalunya 35, 43002 Tarragona, Spain
andreas.maletti@urv.cat

Abstract. Simulations of weighted tree automata (wta) are considered. It is shown how such simulations can be decomposed into simpler functional and dual functional simulations also called forward and backward simulations. In addition, it is shown in several cases (fields, commutative rings, NOETHERIAN semirings, semiring of natural numbers) that all equivalent wta M and N can be joined by a finite chain of simulations. More precisely, in all mentioned cases there is a single wta that simulates both M and N. Those results immediately yield decidability of equivalence provided that the semiring is finitely (and effectively) presented.

1 Introduction

Weighted tree automata are widely used in applications such as model checking [1] and natural language processing [21]. They finitely represent mappings, called tree series, that assign a weight to each tree. For example, a probabilistic parser would return a tree series that assigns to each parse tree its likelihood. Consequently, several toolkits [20,24,10] implement weighted tree automata.

The notion of simulation that is used in this paper is a generalization of the simulations for unweighted and weighted (finite) string automata of [6,15]. The aim is to relate structurally equivalent automata. The results of [6, Section 9.7] and [22] show that two unweighted string automata (i.e., potentially nondeterministic string automata over the BOOLEAN semiring) are equivalent if and only if they can be connected by a finite chain of relational simulations, and that in fact *functional* and *dual functional* simulations are sufficient. Simulations for weighted string automata (wsa) are called *conjugacies* in [3,4], where it is shown that for all fields, many rings including the ring $\mathbb{Z}$ of integers, and the semiring $\mathbb{N}$ of natural numbers, two wsa are equivalent if and only if they can be connected by a finite chain of simulations. It is also shown that even a finite

⋆ Partially supported by grant no. K 75249 from the *National Foundation of Hungary for Scientific Research* and by the TÁMOP-4.2.2/08/1/2008-0008 program of the *Hungarian National Development Agency*.

⋆⋆ Financially supported by the *Ministerio de Educación y Ciencia* grants JDCI-2007-760 and MTM-2007-63422 and the *European Science Foundation* short-visit grant 2978 in the activity "Automata: from Mathematics to Applications".

M. Domaratzki and K. Salomaa (Eds.): CIAA 2010, LNCS 6482, pp. 321–330, 2011.

chain of functional (*covering*) and dual functional (*co-covering*) simulations is sufficient. The origin of those results can be traced back to the pioneering work of Schützenberger in the early 60's, who proved that every wsa over a field is equivalent to a minimal wsa that is simulated by every *trim* equivalent wsa [5]. Relational simulations of wsa are also studied in [9], where they are used to reduce the size of wsa. The relationship between functional simulations and the Milner-Park notion of bisimulation [25,26] is discussed in [6,9].

In this contribution, we investigate simulations for weighted (finite) tree automata (wta). Schützenberger's minimization method was extended to wta over fields in [2,8]. In addition, relational and functional simulations for wta are probably first used in [12,13,18]. Moreover, simulations can be generalized to presentations in algebraic theories [6], which seems to cover all mentioned instances. Here, we extend and improve the results of [3,4] to wta. In particular, we show that two wta over a commutative ring, Noetherian semiring, or the semiring $\mathbb{N}$ are equivalent if and only if they are connected by a finite chain of simulations. Moreover, we discuss when the simulations can be replaced by functional and dual functional simulations, which are efficiently computable [18]. Such results are important because they immediately yield the decidability of equivalence provided that the semiring is finitely and effectively presented.

2 Preliminaries

The set of nonnegative integers is $\mathbb{N}$. For every $k \in \mathbb{N}$, the set $\{i \in \mathbb{N} \mid 1 \leq i \leq k\}$ is simply denoted by $[k]$. We write $|A|$ for the cardinality of the set A. A *semiring* is an algebraic structure $\mathcal{A} = (A, +, \cdot, 0, 1)$ such that $(A, +, 0)$ and $(A, \cdot, 1)$ are monoids, of which the former is commutative, and $\cdot$ distributes both-sided over finite sums (i.e., $a \cdot 0 = 0 = 0 \cdot a$ for every $a \in A$ and $a \cdot (b + c) = ab + ac$ and $(b+c)\cdot a = ba+ca$ for every $a, b, c \in A$). The semiring $\mathcal{A}$ is *commutative* if $(A, \cdot, 1)$ is commutative. It is a *ring* if there exists an element $-1 \in A$ such that $1+(-1) = 0$. The set U is the set $\{a \in A \mid \exists b \in A\colon ab = 1 = ba\}$ of *(multiplicative) units*. The semiring $\mathcal{A}$ is a *semifield* if $U = A \setminus \{0\}$; i.e., for every $a \in A$ there exists a *multiplicative inverse* $a^{-1} \in A$ such that $aa^{-1} = 1 = a^{-1}a$. A *field* is a semifield that is also a ring. Let $\langle B\rangle_+ = \{b_1 + \cdots + b_n \mid n \in \mathbb{N}, b_1, \ldots, b_n \in B\}$ for every $B \subseteq A$. If $A = \langle B\rangle_+$, then $\mathcal{A}$ is *additively generated by* B. The semiring $\mathcal{A}$ is *equisubtractive* if for every $a_1, a_2, b_1, b_2 \in A$ with $a_1 + b_1 = a_2 + b_2$ there exist $c_1, c_2, d_1, d_2 \in A$ such that (i) $a_1 = c_1 + d_1$, (ii) $b_1 = c_2 + d_2$, (iii) $a_2 = c_1 + c_2$, and (iv) $b_2 = d_1 + d_2$. It is *zero-sum free* (*zero-divisor free*, respectively) if $a + b = 0$ ($a \cdot b = 0$, respectively) implies $0 \in \{a, b\}$ for every $a, b \in A$. Finally, it is *positive* if it is both zero-sum and zero-divisor free. Clearly, any nontrivial (i.e., $0 \neq 1$) ring is not zero-sum free, and any semifield is zero-divisor free. An infinitary sum operation $\sum$ is a family $(\sum_I)_I$ such that $\sum_I \colon A^I \to A$. We generally write $\sum_{i\in I} a_i$ instead of $\sum_I (a_i)_{i\in I}$. The semiring $\mathcal{A}$ together with the infinitary sum operation $\sum$ is *complete* [11,17,19] if for all index sets I and $(a_i)_{i\in I} \in A^I$

- $\sum_{i\in I} a_i = a_{j_1} + a_{j_2}$ if $I = \{j_1, j_2\}$ with $j_1 \neq j_2$,
- $\sum_{i\in I} a_i = \sum_{j\in J} \bigl(\sum_{i\in I_j} a_i\bigr)$ for every partition $(I_j)_{j\in J}$ of I, and

– $a \cdot \left(\sum_{i\in I} a_i\right) = \sum_{i\in I} aa_i$ and $\left(\sum_{i\in I} a_i\right) \cdot a = \sum_{i\in I} a_i a$ for every $a \in A$.

An *$\mathcal{A}$-semimodule* is a commutative monoid $(B, +, 0)$ together with an action $\cdot\colon A\times B \to B$, written as juxtaposition, such that for every $a, a' \in A$ and $b, b' \in B$ we have (i) $(a + a')b = ab + a'b$, (ii) $a(b + b') = ab + ab'$, (iii) $0b = 0$, (iv) $1b = b$, and (v) $(a \cdot a')b = a(a'b)$. The semiring $\mathcal{A}$ is NOETHERIAN if all subsemimodules of every finitely-generated $\mathcal{A}$-semimodule are again finitely-generated.

In the following, we identify index sets of equal cardinality. Let $X \in A^{I_1\times J_1}$ and $Y \in A^{I_2\times J_2}$ for finite sets I_1, I_2, J_1, J_2. We use upper-case letters (like C, D, E, X, Y) for matrices and the corresponding lower-case letters for their entries. The matrix X is *relational* if $X \in \{0, 1\}^{I_1\times J_1}$. Clearly, a relational X corresponds to a relation $\rho_X \subseteq I_1 \times J_1$ (and vice versa) by $(i, j) \in \rho_X$ if and only if $x_{ij} = 1$. Moreover, a relational matrix X is *functional*, *surjective*, or *injective* if ρ_X has this property. As usual, we denote the *transpose* of X by X^{T}, and we call X *nondegenerate* if it has no rows or columns of entirely zeroes. A *diagonal* matrix X is such that $x_{ij} = 0$ for every $i \neq j$. Finally, the matrix X is invertible if there exists a matrix X^{-1} such that $XX^{-1} = I = X^{-1}X$ where I is the unit matrix. The KRONECKER product $X \otimes Y \in A^{(I_1\times I_2)\times(J_1\times J_2)}$ is such that $(X \otimes Y)_{i_1 i_2, j_1 j_2} = x_{i_1 j_1} y_{i_2 j_2}$ for every $i_1 \in I_1$, $i_2 \in I_2$, $j_1 \in J_1$, and $j_2 \in J_2$. We let $X^{0,\otimes} = (1)$ and $X^{i+1,\otimes} = X^{i,\otimes} \otimes X$ for every $i \in \mathbb{N}$.

Finally, let us move to trees. A *ranked alphabet* is a finite set Σ together with a mapping $\mathrm{rk}\colon \Sigma \to \mathbb{N}$. We often just write Σ for a ranked alphabet and assume that the mapping rk is implicit. We write $\Sigma_k = \{\sigma \in \Sigma \mid \mathrm{rk}(\sigma) = k\}$ for the set of all k-ary symbols. The set of *Σ-trees* is the smallest set T_Σ such that $\sigma(t_1, \ldots, t_k) \in T_\Sigma$ for all $\sigma \in \Sigma_k$ and $t_1, \ldots, t_k \in T_\Sigma$. A *tree series* is a mapping $\varphi\colon T_\Sigma \to A$. The set of all such tree series is denoted by $A\langle\langle T_\Sigma\rangle\rangle$. For every $\varphi \in A\langle\langle T_\Sigma\rangle\rangle$ and $t \in T_\Sigma$, we often write (φ, t) instead of $\varphi(t)$.

A *weighted tree automaton (over $\mathcal{A}$)*, for short: wta, is a system (Σ, Q, μ, F) with an input ranked alphabet Σ, a finite set Q of *states*, transitions $\mu = (\mu_k)_{k\in\mathbb{N}}$ such that $\mu_k\colon \Sigma_k \to A^{Q^k\times Q}$ for every $k \in \mathbb{N}$, and a *final weight* vector $F \in A^Q$. Next, let us introduce the semantics $\|M\|$ of the wta M. We first define the mapping $h_\mu\colon T_\Sigma \to A^Q$ for every $\sigma \in \Sigma_k$ and $t_1, \ldots, t_k \in T_\Sigma$ by $h_\mu(\sigma(t_1, \ldots, t_k)) = \big(h_\mu(t_1) \otimes \cdots \otimes h_\mu(t_k)\big) \cdot \mu_k(\sigma)$, where the final product $\cdot$ is the classical matrix product. Then $(\|M\|, t) = h_\mu(t)F$ for every $t \in T_\Sigma$, where the product is the usual inner (dot) product. The wta M is *trim* if every state is accessible and co-accessible in the BOOLEAN wta obtained by replacing every nonzero weight by 1.

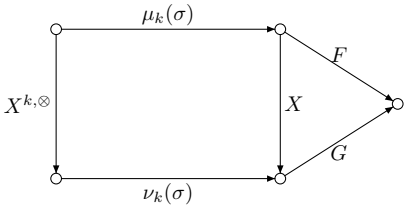


Fig. 1. Illustration of simulation

3 Simulation

Let us introduce the main notion of the paper. From now on, let $M = (\Sigma, Q, \mu, F)$ and $N = (\Sigma, P, \nu, G)$ be wta. Then M *simulates* N (cf., [6,15], [3, Def. 1], and

[12, Def. 35]) if there is $X \in A^{Q \times P}$ such that $F = XG$ and $\mu_k(\sigma)X = X^{k,\otimes} \cdot \nu_k(\sigma)$ for every $\sigma \in \Sigma_k$. The matrix X is called *transfer matrix*, and we write $M \to_X N$ if M simulates N with transfer matrix X. Note that $X^{k,\otimes}_{i_1 \cdots i_k, j_1 \cdots j_k} = \prod_{\ell=1}^{k} x_{i_\ell, j_\ell}$. We illustrate the notion of simulation in Fig. 1. If $M \to_X M'$ and $M' \to_Y N$, then $M \to_{XY} N$. Moreover, if $M \to_X N$, then M and N are equivalent.

Theorem 1. *If M simulates N, then M and N are equivalent.*

Next, we prepare the result for functional simulations. To this end, we first need to prove in which cases the transfer matrix is nondegenerate.

Lemma 2. *Let M and N be trim and $M \to_X N$. If (i) X is functional or (ii) $\mathcal{A}$ is positive, then X is nondegenerate.*

Now we relate functional simulation to forward simulation [18, Def. 1]. A surjective mapping $\rho\colon Q \to P$ is a *forward simulation* from M to N if (i) $F_q = G_{\rho(q)}$ for every $q \in Q$ and (ii) $\sum_{q \in \rho^{-1}(p)} \mu_k(\sigma)_{q_1 \cdots q_k, q} = \nu_k(\sigma)_{\rho(q_1) \cdots \rho(q_k), p}$ for every $p \in P$, $\sigma \in \Sigma_k$, and $q_1, \ldots, q_k \in Q$. We say that M *forward simulates* N, written $M \twoheadrightarrow N$, if there exists a forward simulation from M to N. Similarly, we can relate backward simulation [18, Def. 16] to dual functional simulation. A surjective function $\rho\colon Q \to P$ is a *backward simulation* from M to N if $\sum_{q \in \rho^{-1}(p)} F_q = G_p$ for every $p \in P$ and $\sum_{q_1 \in \rho^{-1}(p_1), \ldots, q_k \in \rho^{-1}(p_k)} \mu_k(\sigma)_{q_1 \cdots q_k, q} = \nu_k(\sigma)_{p_1 \cdots p_k, \rho(q)}$ for every $q \in Q$, $\sigma \in \Sigma_k$, and $p_1, \ldots, p_k \in P$. We say that M *backward simulates* N, written $M \twoheadleftarrow N$, if there exists a backward simulation from M to N. Using Lemma 2 we obtain the following statement.

Lemma 3. *Let N be trim. Then $M \twoheadrightarrow N$ if and only if there exists a functional transfer matrix X such that $M \to_X N$. Moreover, $M \twoheadleftarrow N$ if and only if there exists a transfer matrix X such that X^{T} is functional and $N \to_X M$.*

Next, we recall two important matrix decomposition results of [3].

Lemma 4. *If $A = \langle U \rangle_+$, then for every $X \in A^{Q \times P}$ there exist matrices C, E, D such that (i) $X = CED$, (ii) C^{T} and D are functional, and (iii) E is an invertible diagonal matrix. If (a) X is nondegenerate or (b) $\mathcal{A}$ has (nontrivial) zero-sums, then C^{T} and D can be chosen to be surjective.*

Lemma 5. *Let $\mathcal{A}$ be equisubtractive. Moreover, let $R \in A^Q$ and $C \in A^P$ be such that $\sum_{q \in Q} r_q = \sum_{p \in P} c_p$. Then there exists a matrix $X \in A^{Q \times P}$ with row sums R and column sums C.*

Using all the previous results, we can now obtain the main result of this section, which shows how we can decompose simulation into functional and dual functional simulation (or: forward and backward simulation, respectively).

Theorem 6. *Let $\mathcal{A}$ be equisubtractive with $A = \langle U \rangle_+$. Then $M \to_X N$ if and only if there exist two wta M' and N' such that (i) $M \to_C M'$ where C^{T} is functional, (ii) $M' \to_E N'$ where E is an invertible diagonal matrix, and (iii) $N' \to_D N$ where D is functional. If M and N are trim, then $M' \twoheadleftarrow M$ and $N' \twoheadrightarrow N$.*

Proof. Clearly, $M \to_C M' \to_E N' \to_D N$, which proves that $M \to_{CED} N$. For the converse, Lemma 4 shows that there exist matrices C, E, and D such that $X = CED$, C^{T} and D are functional matrices, and $E \in A^{I \times I}$ is an invertible diagonal matrix. Finally, let $\varphi\colon I \to Q$ and $\psi\colon I \to P$ be the functions associated to C^{T} and D.

It remains to determine the wta M' and N'. Let $M' = (\Sigma, I, \mu', F')$ and $N' = (\Sigma, I, \nu', G')$ with $G' = DG$ and $F' = EDG$. Then we have $CF' = CEDG = XG = F$. Thus, it remains to specify $\mu'_k(\sigma)$ and $\nu'_k(\sigma)$ for every $\sigma \in \Sigma_k$. To this end, we determine a matrix $Y \in A^{I^k \times I}$ such that we have (1) $C^{k,\otimes} \cdot Y = \mu_k(\sigma)CE$ and (2) $YD = E^{k,\otimes} \cdot D^{k,\otimes} \cdot \nu_k(\sigma)$. Let $\mu'_k(\sigma) = YE^{-1}$ and $\nu'_k(\sigma) = (E^{k,\otimes})^{-1} \cdot Y$. Consequently, we have $\mu_k(\sigma)C = C^{k,\otimes} \cdot \mu'_k(\sigma)$, $\mu'_k(\sigma)E = E^{k,\otimes} \cdot \nu'_k(\sigma)$, and $\nu'_k(\sigma)D = D^{k,\otimes} \cdot \nu_k(\sigma)$. These equalities are displayed in Fig. 2 (right).

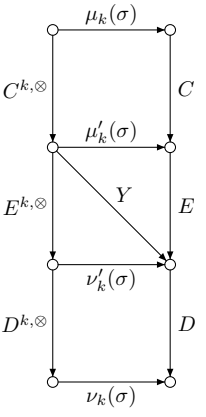


Fig. 2. Relations between the matrices

Finally, we need to specify the matrix Y. For every $q \in Q$ and $p \in P$, let $I_q = \varphi^{-1}(q)$ and $J_p = \psi^{-1}(p)$. Obviously, Y can be decomposed into disjoint (not necessarily contiguous) submatrices $Y_{q_1 \cdots q_k, p} \in A^{(I_{q_1} \times \cdots \times I_{q_k}) \times J_p}$ with $q_1, \ldots, q_k \in Q$ and $p \in P$. Then properties (1) and (2) hold if and only if for every $q_1, \ldots, q_k \in Q$ and $p \in P$ the following two conditions hold:

1. For every $i \in I$ such that $\psi(i) = p$, the sum of the i-column of $Y_{q_1 \cdots q_k, p}$ is $\mu_k(\sigma)_{q_1 \cdots q_k, \varphi(i)} \cdot e_{i,i}$.
2. For all $i_1, \ldots, i_k \in I$ such that $\varphi(i_j) = q_j$ for every $j \in [k]$, the sum of the $(i_1, \ldots, i_k)$-row of $Y_{q_1 \cdots q_k, p}$ is $\prod_{j=1}^{k} e_{i_j, i_j} \cdot \nu_k(\sigma)_{\psi(i_1) \cdots \psi(i_k), p}$.

Those two conditions are compatible because

$$\sum_{\substack{i \in I \\ \psi(i) = p}} \mu_k(\sigma)_{q_1 \cdots q_k, \varphi(i)} \cdot e_{i,i} = \big(\mu_k(\sigma)CED\big)_{q_1 \cdots q_k, p} = \big(\mu_k(\sigma)X\big)_{q_1 \cdots q_k, p}$$

$$= \big(X^{k,\otimes} \cdot \nu_k(\sigma)\big)_{q_1 \cdots q_k, p} = \big(C^{k,\otimes} \cdot E^{k,\otimes} \cdot D^{k,\otimes} \cdot \nu_k(\sigma)\big)_{q_1 \cdots q_k, p}$$

$$= \sum_{\substack{i_1, \ldots, i_k \in I \\ \forall j \in [k]\colon\, \varphi(i_j) = q_j}} \Big(\prod_{j=1}^{k} e_{i_j, i_j}\Big) \cdot \nu_k(\sigma)_{\psi(i_1) \cdots \psi(i_k), p} \ .$$

Consequently, the row and column sums of the submatrices $Y_{q_1 \cdots q_k, p}$ are consistent, which yields that we can determine all the submatrices (and thus the whole matrix) by Lemma 5. If M and N are trim, then either (a) $\mathcal{A}$ is zero-sum free (and thus positive because it is additively generated by its units), in which case X is nondegenerate by Lemma 2, or (b) $\mathcal{A}$ has nontrivial zero-sums. In both cases, Lemma 4 shows that the matrices C^{T} and D are surjective, which yields the additional statement by Lemma 3. □

The decomposition of simulations into forward and backward simulation is effective and has computational benefits because it is shown in [18] that forward and backward simulation can be efficiently computed. To keep the presentation simple, we will continue to deal with simulation in the following. However, in many of the following cases they can be decomposed.

4 Category of Simulations

In this section our aim is to show that several well-known constructions of wta are *functorial*: they may be extended to simulations in a functorial way. Below we will only deal with the sum, HADAMARD product, σ_0-product, and σ_0-iteration (cf. [14]). Scalar OI-substition, † (the dagger operation) [7], homomorphism, quotient, and top-concatenation [14] may be covered in a similar fashion.

In this section, let $\mathcal{A}$ be commutative. Moreover, let $M = (\Sigma, Q, \mu, F)$, $M' = (\Sigma, Q', \mu', F')$, and $M'' = (\Sigma, Q'', \mu'', F'')$ be wta. We already remarked that, if $M \to_X M'$ and $M' \to_Y M''$, then $M \to_{XY} M''$. Moreover, $M \to_I M$ with the unit matrix $I \in A^{Q \times Q}$. Thus, wta over the alphabet Σ form a category $\mathbf{Sim}_\Sigma$.

In the following, let $M = (\Sigma, Q, \mu, F)$ and $N = (\Sigma, P, \nu, G)$ be wta such that $Q \cap P = \emptyset$. The sum $M + N$ of M and N is the wta $(\Sigma, Q \cup P, \kappa, H)$ where $H = \langle F, G \rangle$ and

$$\kappa_k(\sigma)_{q_1 \cdots q_k, q} = (\mu_k(\sigma) + \nu_k(\sigma))_{q_1 \cdots q_k, q} = \begin{cases} \mu_k(\sigma)_{q_1 \cdots q_k, q} & \text{if } q, q_1, \ldots, q_k \in Q \\ \nu_k(\sigma)_{q_1 \cdots q_k, q} & \text{if } q, q_1, \ldots, q_k \in P \\ 0 & \text{otherwise.} \end{cases}$$

for all $\sigma \in \Sigma_k$ and $q, q_1, \ldots, q_k \in Q \cup P$. It is known that $\|M + N\| = \|M\| + \|N\|$. Next, we extend the sum construction to simulations. To this end, let $M \to_X M'$ and $N \to_Y N'$ with $N' = (\Sigma, P', \nu', G')$. The sum $X + Y \in A^{(Q \cup P) \times (Q' \cup P')}$ of the transfer matrices X and Y is $X + Y = \left(\begin{smallmatrix} X & 0 \\ 0 & Y \end{smallmatrix}\right)$. Then $(M + N) \to_{X+Y} (M' + N')$.

Proposition 7. *The function $+$, which is defined on wta and transfer matrices, is a functor $\mathbf{Sim}_\Sigma^2 \to \mathbf{Sim}_\Sigma$.*

Next, we treat the remaining operations. Let σ_0 be a distinguished symbol in Σ_0. The σ_0-product $M \cdot_{\sigma_0} N$ of M with N is the wta $(\Sigma, Q \cup P, \kappa, H)$ such that $H = \langle F, 0 \rangle$ and for each $\sigma \in \Sigma_k$ with $\sigma \neq \sigma_0$,

$$\kappa_k(\sigma)_{q_1 \cdots q_k, q} = \begin{cases} \mu_k(\sigma)_{q_1 \cdots q_k, q} & \text{if } q, q_1, \ldots, q_k \in Q \\ \mu_0(\sigma_0)_q \cdot \sum_{p \in P} \nu_k(\sigma)_{q_1 \cdots q_k, p} G_p & \text{if } q \in Q \text{ and } q_1, \ldots, q_k \in P \\ \nu_k(\sigma)_{q_1 \cdots q_k, q} & \text{if } q, q_1, \ldots, q_k \in P \\ 0 & \text{otherwise.} \end{cases}$$

Moreover,

$$\kappa_0(\sigma_0)_q = \begin{cases} \mu_0(\sigma_0)_q \cdot \sum_{p \in P} \nu_0(\sigma_0)_p G_p & \text{if } q \in Q \\ \nu_0(\sigma_0)_q & \text{if } q \in P. \end{cases}$$

It is known that $\|M \cdot_{\sigma_0} N\| = \|M\| \cdot_{\sigma_0} \|N\|$. Let $M \to_X M'$ and $N \to_Y N'$. We define $X \cdot_{\sigma_0} Y = X + Y$. The HADAMARD product $M \cdot_{\mathrm{H}} N$ is the wta $(\Sigma, Q \times P, \kappa, H)$ where $H = F \otimes G$ and $\kappa_k(\sigma) = \mu_k(\sigma) \otimes \nu_k(\sigma)$ for all $\sigma \in \Sigma_k$. If $M \to_X M'$ and $N \to_Y N'$, then we define $X \cdot_{\mathrm{H}} Y = X \otimes Y$. Finally, let $\mathcal{A}$ be complete. Thus, $\mathcal{A}$ allows the definition of the star operation $a^* = \sum_{n \in \mathbb{N}} a^n$ for every $a \in A$. The σ_0-iteration $M^{*\sigma_0}$ of M is the wta (Σ, Q, κ, F) where

$$\kappa_k(\sigma)_{q_1 \cdots q_k, q} = \mu_k(\sigma)_{q_1 \cdots q_k, q} + \|M\|(\sigma_0)^* \cdot \sum_{p \in Q} \mu_k(\sigma)_{q_1 \cdots q_k, p} F_p$$

for all $\sigma \in \Sigma_k \setminus \{\sigma_0\}$ and $\kappa_0(\sigma_0) = \mu_0(\sigma_0)$. If $M \to_X M'$, then we define $X^{*\sigma_0} = X$.

Proposition 8. *The functions $\cdot_{\sigma_0}$ and $\cdot_{\mathrm{H}}$, which are defined on wta and transfer matrices, are functors $\mathbf{Sim}_\Sigma^2 \to \mathbf{Sim}_\Sigma$. Moreover, σ_0-iteration is a functor $\mathbf{Sim}_\Sigma \to \mathbf{Sim}_\Sigma$ if $\mathcal{A}$ is complete.*

5 Joint Reduction

In this section, we will establish equivalence results using an improved version of the approach called *joint reduction* in [4]. Let $V \subseteq A^I$ be a set of vectors for a finite set I. The $\mathcal{A}$-semimodule generated by V is denoted by $\langle V \rangle$. Given two wta $M = (\Sigma, Q, \mu, F)$ and $N = (\Sigma, P, \nu, G)$ with $Q \cap P = \emptyset$, we first compute $M + N = (\Sigma, Q \cup P, \mu', F')$ as defined in Section 4. The aim is to compute a finite set $V \subseteq A^{Q \cup P}$ such that

(i) $(v_1 \otimes \cdots \otimes v_k) \cdot \mu'_k(\sigma) \in \langle V \rangle$ for every $\sigma \in \Sigma_k$ and $v_1, \ldots, v_k \in V$, and
(ii) $v_1 F = v_2 G$ for every $(v_1, v_2) \in V$ such that $v_1 \in A^Q$ and $v_2 \in A^P$.

With such a finite set V we can now construct a wta $M' = (\Sigma, V, \nu', G')$ with $G'_v = vF'$ for every $v \in V$ and $\sum_{v \in V} \nu'_k(\sigma)_{v_1 \cdots v_k, v} \cdot v = (v_1 \otimes \cdots \otimes v_k) \cdot \mu'_k(\sigma)$ for every $\sigma \in \Sigma_k$ and $v_1, \ldots, v_k \in V$. It remains to prove that M' simulates $M + N$. To this end, let $X = (v)_{v \in V}$ where each $v \in V$ is a row vector. Then for every $\sigma \in \Sigma_k$, $v_1, \ldots, v_k \in V$, and $q \in Q \cup P$, we have

$$\begin{aligned}
(\nu'_k(\sigma) X)_{v_1 \cdots v_k, q} &= \sum_{v \in V} \nu'_k(\sigma)_{v_1 \cdots v_k, v} \cdot v_q = \Big(\sum_{v \in V} \nu'_k(\sigma)_{v_1 \cdots v_k, v} \cdot v \Big)_q \\
&= \big((v_1 \otimes \cdots \otimes v_k) \cdot \mu'_k(\sigma) \big)_q = \sum_{q_1, \ldots, q_k \in Q \cup P} (v_1)_{q_1} \cdot \ldots \cdot (v_k)_{q_k} \cdot \mu'_k(\sigma)_{q_1 \cdots q_k, q} \\
&= \big(X^{k, \otimes} \cdot \mu'_k(\sigma) \big)_{v_1 \cdots v_k, q} \ .
\end{aligned}$$

Moreover, if we let X_1 and X_2 be the restrictions of X to the entries of Q and P, respectively, then we have $\nu'_k(\sigma) X_1 = X_1^{k, \otimes} \cdot \mu_k(\sigma)$ and $\nu'_k(\sigma) X_2 = X_2^{k, \otimes} \cdot \nu_k(\sigma)$. In addition, $G'_v = vF' = \sum_{q \in Q \cup P} v_q F'_q = (XF')_v$ for every $v \in V$, which proves that $M' \to_X (M + N)$. Since $v_1 F = v_2 G$ for every $(v_1, v_2) \in V$, we

have $G'_{(v_1,v_2)} = (v_1, v_2)F' = v_1F + v_2G = (1+1)v_1F = (1+1)v_2G$. Now, let $G''_{(v_1,v_2)} = v_1F = v_2G$ for every $(v_1, v_2) \in V$. Then

$$(X_2G)_v = \sum_{p \in P} v_p G_p = v_2G = G''_v = v_1F = \sum_{q \in Q} v_q F_q = (X_1F)_v$$

for every $v = (v_1, v_2) \in V$. Consequently, $M'' \to_{X_1} M$ and $M'' \to_{X_2} N$, where $M'' = (\Sigma, V, \nu', G'')$. This proves the next theorem.

Theorem 9. *Let M and N be equivalent. If there exists a finite set $V \subseteq A^{Q \cup P}$ with properties (i) and (ii), then a finite chain of simulations joins M and N. In fact, there exists a single wta that simulates both M and N.*

Let us first recall a known result [2] for fields. Note that, in comparison to our results, the single wta can be chosen to be a minimal wta.

Theorem 10 (see [2, p. 453]). *Every two equivalent trim wta M and N over a field $\mathcal{A}$ can be joined by a finite chain of simulations. Moreover, there exists a minimal wta that simulates both M and N.*

We can obtain a similar theorem with the help of Theorem 9 as follows. Let $\mathcal{A}$ be a NOETHERIAN semiring. Let $V_0 = \{\mu'_0(\alpha) \mid \alpha \in \Sigma_0\}$ and

$$V_{i+1} = V_i \cup \big(\{(v_1 \otimes \cdots \otimes v_k) \cdot \mu'_k(\sigma) \mid \sigma \in \Sigma_k, v_1, \ldots, v_k \in V_i\} \setminus \langle V_i \rangle\big)$$

for every $i \in \mathbb{N}$. Then $\{0\} \subseteq \langle V_0 \rangle \subseteq \langle V_1 \rangle \subseteq \cdots \subseteq \langle V_k \rangle \subseteq \cdots$ is stationary after finitely many steps because $\mathcal{A}$ is NOETHERIAN. Thus, let $V = V_k$ for some $k \in \mathbb{N}$ such that $\langle V_k \rangle = \langle V_{k+1} \rangle$. Clearly, V is finite and has property (i). Trivially, $V \subseteq \{h_{\mu'}(t) \mid t \in T_\Sigma\}$, so let $v \in V$ be such that $v = \sum_{i \in I}(h_\mu(t_i), h_\nu(t_i))$ for some finite index set I and $t_i \in T_\Sigma$ for every $i \in I$. Then

$$\Big(\sum_{i \in I} h_\mu(t_i)\Big)F = \sum_{i \in I}(\|M\|, t_i) = \sum_{i \in I}(\|N\|, t_i) = \Big(\sum_{i \in I} h_\nu(t_i)\Big)G$$

because $\|M\| = \|N\|$, which proves property (ii).

In fact, since $M+N$ uses only finitely many semiring coefficients, it is sufficient that every finitely generated subsemiring of $\mathcal{A}$ is contained in a NOETHERIAN subsemiring of $\mathcal{A}$. Then the following theorem follows from Theorem 9.

Theorem 11. *Let $\mathcal{A}$ be such that every finitely generated subsemiring is contained in a* NOETHERIAN *subsemiring of $\mathcal{A}$. For all equivalent wta M and N over $\mathcal{A}$, there exists a finite chain of simulations that join M and N. In fact, there exists a single wta that simulates both M and N.*

Note that $\mathbb{Z}$ is a NOETHERIAN ring. More generally, every finitely generated commutative ring is NOETHERIAN [23, Cor. IV.2.4 & Prop. X.1.4].

Corollary 12 (of Theorem 11). *For all equivalent wta M and N over a commutative ring $\mathcal{A}$, there exists a finite chain of simulations that join M and N. In fact, there exists a single wta that simulates both M and N.*

Finally, let $\mathcal{A} = \mathbb{N}$ be the semiring of natural numbers. We compute the finite set $V \subseteq \mathbb{N}^{Q \cup P}$ as follows:

1. Let $V_0 = \{\mu'_0(\alpha) \mid \alpha \in \Sigma_0\}$ and $i = 0$.
2. For every $v, v' \in V_i$ such that $v \leq v'$, replace v' by $v' - v$.
3. Set $V_{i+1} = V_i \cup \big(\{(v_1 \otimes \cdots \otimes v_k) \cdot \mu'_k(\sigma) \mid \sigma \in \Sigma_k, v_1, \ldots, v_k \in V_i\} \setminus \langle V_i \rangle\big)$.
4. Until $V_{i+1} = V_i$, increase i and repeat step 2.

Clearly, this algorithm terminates since every vector can only be replaced by a smaller vector in step 2 and step 3 only adds a finite number of vectors, which after the reduction in step 2 are pairwise incomparable. Moreover, property (i) trivially holds because at termination $V_{i+1} = V_i$ after step 3. Consequently, we only need to prove property (ii). To this end, we first prove that $V \subseteq \langle \{h_{\mu'}(t) \mid t \in T_\Sigma\} \rangle_{+,-}$. This is trivially true after step 1 because $\mu'_0(\alpha) = h_{\mu'}(\alpha)$ for every $\alpha \in \Sigma_0$. Clearly, the property is preserved in steps 2 and 3. Finally, property (ii) can now be proved as follows. Let $v \in V$ be such that $v = \sum_{i \in I_1} (h_\mu(t_i), h_\nu(t_i)) - \sum_{i \in I_2} (h_\mu(t_i), h_\nu(t_i))$ for some finite index sets I_1 and I_2 and $t_i \in T_\Sigma$ for every $i \in I_1 \cup I_2$. Then by $\|M\| = \|N\|$ we obtain

$$\begin{aligned}
&\Big(\sum_{i \in I_1} h_\mu(t_i) - \sum_{i \in I_2} h_\mu(t_i)\Big) F = \sum_{i \in I_1} h_\mu(t_i) F - \sum_{i \in I_2} h_\mu(t_i) F \\
&= \sum_{i \in I_1} (\|M\|, t_i) - \sum_{i \in I_2} (\|M\|, t_i) = \sum_{i \in I_1} (\|N\|, t_i) - \sum_{i \in I_2} (\|N\|, t_i) \\
&= \sum_{i \in I_1} h_\nu(t_i) G - \sum_{i \in I_2} h_\nu(t_i) G = \Big(\sum_{i \in I_1} h_\nu(t_i) - \sum_{i \in I_2} h_\nu(t_i)\Big) G \ .
\end{aligned}$$

Corollary 13 (of Theorem 9). *For all equivalent wta M and N over $\mathbb{N}$, there exists a finite chain of simulations that join M and N. In fact, there exists a single wta that simulates both M and N.*

For all finitely and effectively presented semirings, Theorems 10 and 11 and Corollaries 12 and 13 also yield decidability of equivalence for M and N. Essentially, we run the trivial semi-decidability test for inequality and a search for the wta that simulates both M and N in parallel. We know that either test will eventually return, thus deciding whether M and N are equivalent. Conversely, if equivalence is undecidable, then simulation cannot capture equivalence [16].

References

1. Abdulla, P.A., Jonsson, B., Mahata, P., d'Orso, J.: Regular tree model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 555–568. Springer, Heidelberg (2002)
2. Alexandrakis, A., Bozapalidis, S.: Représentations matricielles des séries d'arbre reconnaissables. Informatique Théorique et Applications 23(4), 449–459 (1989)
3. Béal, M.P., Lombardy, S., Sakarovitch, J.: On the equivalence of $\mathbb{Z}$-automata. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 397–409. Springer, Heidelberg (2005)

4. Béal, M.P., Lombardy, S., Sakarovitch, J.: Conjugacy and equivalence of weighted automata and functional transducers. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 58–69. Springer, Heidelberg (2006)
5. Berstel, J., Reutenauer, C.: Rational Series and Their Languages. EATCS Monographs on Theoret. Comput. Sci., vol. 12. Springer, Heidelberg (1984)
6. Bloom, S.L., Ésik, Z.: Iteration theories: The Equational Logic of Iterative Processes. Springer, Heidelberg (1993)
7. Bloom, S.L., Ésik, Z.: An extension theorem with an application to formal tree series. J. Autom. Lang. Combin. 8(2), 145–185 (2003)
8. Bozapalidis, S.: Effective construction of the syntactic algebra of a recognizable series on trees. Acta Inform. 28(4), 351–363 (1991)
9. Buchholz, P.: Bisimulation relations for weighted automata. Theoret. Comput. Sci. 393(1-3), 109–123 (2008)
10. Cleophas, L.: Forest FIRE and FIRE wood: Tools for tree automata and tree algorithms. In: FSMNLP, pp. 191–198 (2008)
11. Eilenberg, S.: Automata, Languages, and Machines. Academic Press, London (1974)
12. Ésik, Z.: Axiomatizing the equational theory of regular tree languages. In: Meinel, C., Morvan, M. (eds.) STACS 1998. LNCS, vol. 1373, pp. 455–465. Springer, Heidelberg (1998)
13. Ésik, Z.: Axiomatizing the equational theory of regular tree languages. J. Log. Algebr. Program. 79(2), 189–213 (2010)
14. Ésik, Z.: Fixed point theory. In: Handbook of Weighted Automata. EATCS Monographs on Theoret. Comput. Sci., pp. 29–66. Springer, Heidelberg (2010)
15. Ésik, Z., Kuich, W.: A generation of Kozen's axiomatization of the equational theory of the regular sets. In: Words, Semigroups, and Transductions, pp. 99–114. World Scientific, Singapore (2001)
16. Ésik, Z., Maletti, A.: Simulation vs. equivalence. In: FCS, pp. 119–122. CSREA Press (2010), preprint: http://arxiv.org/abs/1004.2426
17. Hebisch, U., Weinert, H.J.: Semirings—Algebraic Theory and Applications in Computer Science. World Scientific, Singapore (1998)
18. Högberg, J., Maletti, A., May, J.: Bisimulation minimisation for weighted tree automata. In: Harju, T., Karhumäki, J., Lepistö, A. (eds.) DLT 2007. LNCS, vol. 4588, pp. 229–241. Springer, Heidelberg (2007)
19. Karner, G.: Continuous monoids and semirings. Theoret. Comput. Sci. 318(3), 355–372 (2004)
20. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual (2001)
21. Knight, K., Graehl, J.: An overview of probabilistic tree transducers for natural language processing. In: Gelbukh, A. (ed.) CICLing 2005. LNCS, vol. 3406, pp. 1–24. Springer, Heidelberg (2005)
22. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. Inform. and Comput. 110(2), 366–390 (1994)
23. Lang, S.: Algebra, 2nd edn. Addison Wesley, Reading (1984)
24. May, J., Knight, K.: TIBURON: A weighted tree automata toolkit. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 102–113. Springer, Heidelberg (2006)
25. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1980)
26. Park, D.M.R.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)

Author Index