Adrian-Horia Dediu
Carlos Martín-Vide
Bianca Truthe (Eds.)

LNCS 7810

Language and Automata Theory and Applications

7th International Conference, LATA 2013
Bilbao, Spain, April 2013
Proceedings

Springer

Adrian-Horia Dediu Carlos Martín-Vide
Bianca Truthe (Eds.)

Language and Automata Theory and Applications

7th International Conference, LATA 2013
Bilbao, Spain, April 2-5, 2013
Proceedings

Springer

Volume Editors

Adrian-Horia Dediu
Universitat Rovira i Virgili, Research Group on Mathematical Linguistics
Avinguda Catalunya, 35, 43002 Tarragona, Spain
E-mail: adrian.dediu@urv.cat

Carlos Martín-Vide
Universitat Rovira i Virgili, Research Group on Mathematical Linguistics
Avinguda Catalunya, 35, 43002 Tarragona, Spain
E-mail: carlos.martin@urv.cat

Bianca Truthe
Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik
Institut für Wissens- und Sprachverarbeitung
Universitätsplatz 2, 39106 Magdeburg, Germany
E-mail: truthe@iws.cs.uni-magdeburg.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-37063-2 e-ISBN 978-3-642-37064-9
DOI 10.1007/978-3-642-37064-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013932557

CR Subject Classification (1998): F.1, F.4, F.2-3, I.2, I.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

These proceedings contain the papers that were presented at the 7th International Conference on Language and Automata Theory and Applications (LATA 2013), held in Bilbao, Spain, during April 2-5, 2013.

The scope of LATA is rather broad, including: algebraic language theory; algorithms for semi-structured data mining; algorithms on automata and words; automata and logic; automata for system analysis and program verification; automata, concurrency and Petri nets; automatic structures; cellular automata; combinatorics on words; computability; computational complexity; computational linguistics; data and image compression; decidability questions on words and languages; descriptional complexity; DNA and other models of bio-inspired computing; document engineering; foundations of finite state technology; foundations of XML; fuzzy and rough languages; grammars (Chomsky hierarchy, contextual, multidimensional, unification, categorial, etc.); grammars and automata architectures; grammatical inference and algorithmic learning; graphs and graph transformation; language varieties and semigroups; language-based cryptography; language-theoretic foundations of artificial intelligence and artificial life; parallel and regulated rewriting; parsing; pattern recognition; patterns and codes; power series; quantum, chemical, and optical computing; semantics; string and combinatorial issues in computational biology and bioinformatics; string processing algorithms; symbolic dynamics; symbolic neural networks; term rewriting; transducers; trees, tree languages and tree automata; weighted automata.

LATA 2013 received 97 submissions. Each one was reviewed by three Program Committee members, many of whom consulted with external referees. After a thorough and vivid discussion phase, the committee decided to accept 45 papers (which represents an acceptance rate of 46.39%). The conference program also included four invited talks and two invited tutorials. Part of the success in the management of such a large number of submissions is due to the excellent facilities provided by the EasyChair conference management system.

We would like to thank all invited speakers and authors for their contributions, the Program Committee and the reviewers for their cooperation, and Springer for its very professional publishing work.

January 2013

Adrian-Horia Dediu
Carlos Martín-Vide
Bianca Truthe

Organization

LATA 2013 was organized by the Basque Center for Applied Mathematics, Bilbao, and the Research Group on Mathematical Linguistics — GRLMC from the University Rovira i Virgili, Tarragona.

Program Committee

Parosh Aziz Abdulla	Uppsala University, Sweden
Franz Baader	Dresden University of Technology, Germany
Jos Baeten	CWI Amsterdam, The Netherlands
Christel Baier	Dresden University of Technology, Germany
Gerth Stølting Brodal	Aarhus University, Denmark
John Case	University of Delaware, Newark, USA
Marek Chrobak	University of California, Riverside, USA
Mariangiola Dezani	University of Turin, Italy
Rod Downey	Victoria University of Wellington, New Zealand
Ding-Zhu Du	University of Texas, Dallas, USA
E. Allen Emerson	University of Texas, Austin, USA
Javier Esparza	Technical University of Munich, Germany
Michael R. Fellows	Charles Darwin University, Darwin, Australia
Alain Finkel	ENS Cachan, France
Dov M. Gabbay	King's College, London, UK
Jürgen Giesl	Aachen University, Germany
Rob van Glabbeek	NICTA, Sydney, Australia
Georg Gottlob	University of Oxford, UK
Annegret Habel	University of Oldenburg, Germany
Reiko Heckel	University of Leicester, UK
Sanjay Jain	National University of Singapore
Charanjit S. Jutla	IBM Thomas J. Watson Research Center, Hawthorne, USA
Ming-Yang Kao	Northwestern University, Evanston, USA
Deepak Kapur	University of New Mexico, Albuquerque, USA
Joost-Pieter Katoen	Aachen University, Germany
S. Rao Kosaraju	Johns Hopkins University, Baltimore, USA
Evangelos Kranakis	Carleton University, Ottawa, Canada
Hans-Jörg Kreowski	University of Bremen, Germany
Tak-Wah Lam	University of Hong Kong, China
Gad M. Landau	University of Haifa, Israel
Kim G. Larsen	Aalborg University, Denmark

Richard Lipton	Georgia Institute of Technology, Atlanta, USA
Jack Lutz	Iowa State University, Ames, USA
Ian Mackie	École Polytechnique, Palaiseau, France
Rupak Majumdar	Max Planck Institute, Kaiserslautern, Germany
Carlos Martín-Vide (Chair)	Rovira i Virgili University, Tarragona, Spain
Paliath Narendran	University at Albany, SUNY, USA
Tobias Nipkow	Technical University of Munich, Germany
David A. Plaisted	University of North Carolina, Chapel Hill, USA
Jean-François Raskin	Free University of Brussels, Belgium
Wolfgang Reisig	Humboldt University, Berlin, Germany
Michaël Rusinowitch	LORIA, Nancy, France
Davide Sangiorgi	University of Bologna, Italy
Bernhard Steffen	Technical University of Dortmund, Germany
Colin Stirling	University of Edinburgh, UK
Alfonso Valencia	CNIO, Madrid, Spain
Helmut Veith	Vienna University of Technology, Austria
Heribert Vollmer	Leibniz University of Hannover, Germany
Osamu Watanabe	Tokyo Institute of Technology, Japan
Pierre Wolper	University of Liège, Belgium
Louxin Zhang	National University of Singapore

External Reviewers

Amit, Mika
Anantharaman, Siva
Atkinson, Mike
Boigelot, Bernard
Bollig, Benedikt
Bonnet, Rémi
Bouchard, Chris
Braibant, Thomas
Carle, Benjamin
Charatonik, Witold
Charlier, Emilie
Chistikov, Dmitry
Dang, Zhe
Davoodi, Pooya
Delzanno, Giorgio
Dubslaff, Clemens
Fates, Nazim
Fernau, Henning
Filiot, Emmanuel
Flick, Nils Erik
Formenti, Enrico
Ganty, Pierre
Gasarch, William
Gentilini, Raffaela
Gero, Kimberly
Gierds, Christian
Haase, Christoph
Harju, Tero
Hertrampf, Ulrich
Holik, Lukas
Holub, Stepan
Howar, Falk
Isberner, Malte
Janicki, Ryszard
Jezequel, Loig
Kejlberg-Rasmussen, Casper
Khoussainov, Bhakhadyr
Klaedtke, Felix
Konnov, Igor
Kopelowitz, Tsvi
Kuester, Jochen
Kuske, Dietrich

Lee, Lap-Kei
Leiß, Hans
Lushman, Brad
Luttenberger, Michael
Makowsky, Johann
Malik, Avinash
Manea, Florin
Marx, Maarten
McKenzie, Pierre
Merten, Maik
Mignosi, Filippo
Murlak, Filip
Müller, Moritz
Müller, Richard
Narayan Kumar, K.
Nguyen, Viet Yen
Niehren, Joachim
Nielsen, Jesper Asbjørn Sindahl
Noll, Thomas
Nowotka, Dirk
Okhotin, Alexander
Oliart, Alberto
Perrin, Dominique
Piskorski, Jakub
Prüfer, Robert
Pulman, Stephen
Rabkin, Max
Radke, Hendrik
Ringeissen, Christophe
Riveros, Cristian
Rosenkrantz, Daniel
Rozenberg, Liat
Rubin, Sasha
Rüthing, Oliver
Saivasan, Prakash
Salvati, Sylvain
Seidl, Helmut
Servais, Frédéric
Servetto, Marco
Shallit, Jeffrey
Siebert, Heike
Sproston, Jeremy
Stephan, Frank
Sürmeli, Jan
Talpin, Jean-Pierre
Tittmann, Peter
Traytel, Dmitriy
Venturini, Rossano
Vialette, Stéphane
Wagner, Christoph
Wedel Vildhøj, Hjalte
Widder, Josef
Wimmer, Ralf
Zuleger, Florian
Yagnatinsky, Mark

Organizing Committee

Adrian-Horia Dediu	Tarragona
Peter Leupold	Tarragona
Carlos Martín-Vide	Tarragona (Co-chair)
Magaly Roldán	Bilbao
Bianca Truthe	Magdeburg
Florentina-Lilica Voicu	Tarragona
Enrique Zuazua	Bilbao (Co-chair)

Table of Contents

Complexity Dichotomy for Counting Problems

Jin-Yi Cai

University of Wisconsin-Madison, WI 53706, USA
jyc@cs.wisc.edu

Abstract. I would like to report on some significant progress in the study of the exact complexity of counting problems. Specifically I will describe the classification program of counting complexity of locally specified problems. This classification program is advanced in three interrelated frameworks: Graph Homomorphisms, Counting CSP, and Holant Problems. In each formulation, complexity dichotomy theorems have been achieved which classify every problem in a given class to be either solvable in polynomial time or #P-hard.

1 Introduction

Computational complexity theory is a branch of Theoretical Computer Science. The primary goal of complexity theory is to classify computational problems according to their inherent computational complexity. The success of this theory is judged by how complete a classification one can obtain. Unfortunately we currently lack any strong lower bound that pertain to the conjectured separation of natural complexity classes such as P vs. NP. For example, we cannot prove any super linear circuit lower bound for any natural problem in NP. Instead, the great success of complexity theory in the past 40 years has been through the notion of completeness, which has the implication that, e.g., assuming $P \neq NP$, then every NP-complete problem is not solvable in polynomial time.

For counting problems the corresponding notion is #P-completeness or more generally #P-hardness. #P is the class of all functions that correspond to counting the number of accepting computations of an NP machine. This class was introduced by Valiant [20] and is the analogue of NP for counting problems. It is stronger than the Polynomial-time Hierarchy [19]. Assuming #P is separate from P, a statement trivially implied by $P \neq NP$, every #P-hard function cannot be computed in polynomial time. As an overly ambitious goal, one might wish to classify *every* function in #P to be either computable in polynomial time or #P-hard. Such a result is called a complexity dichotomy theorem [18]. However for the whole #P this is known to be *false*, assuming #P is separate from P. Therefore, for counting problems, assuming #P is separate from P, one cannot hope to prove such a dichotomy theorem for all of #P. Instead one aims to achieve such a complete classification for as broad a class of problems as possible.

What appears to be the broadest type of problems for which such a complete classification is feasible are the sum-of-product computations. For these problems, recently there has been remarkable progress on this classification program.

Progress has been made in at least three frameworks: Graph Homomorphisms, Counting CSP, and Holant problems. In each case strong and broadly applicable complexity dichotomy theorems have been achieved.

2 Partition Function of Graph Homomorphisms

Graph homomorphism is defined as follows [17,16]. Given a $k \times k$ matrix $\mathbf{A}$, the graph homomorphism function, a.k.a. the *partition function*, $Z_{\mathbf{A}}(G)$ is defined as

$$Z_{\mathbf{A}}(G) = \sum_{\xi : V \to [k]} \prod_{(u,v) \in E} A_{\xi(u),\xi(v)},$$

where $G = (V, E)$ is any input graph. If $\mathbf{A}$ is symmetric and G is an undirected graph, then this is graph homomorphism for undirected graphs, which has been studied most intensively over the years. The function $Z_{\mathbf{A}}(G)$ can encode many interesting graph properties. E.g. counting vertex covers corresponds to the 2 by 2 matrix $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ (this matrix enclodes the binary Or function), and counting k-colorings corresponds to the k by k matrix with 0's on the diagonal and 1's off diagonal (this is the binary DisEquality function on domain size k).

Dyer and Greenhill were the first to give a complexity dichotomy theorem for $Z_{\mathbf{A}}(G)$, where the matrix $\mathbf{A}$ is an arbitrary fixed 0-1 symmetric matrix [12]. Their elegant dichotomy criterion is as follows: Let H be the undirected graph with adjacency matrix $\mathbf{A}$. Then $Z_{\mathbf{A}}(G)$ is computable in polynomial time in the size of G, provided every connected component of H is an isolated vertex without a loop, or a complete graph with all loops present, or a complete unlooped bipartite graph. In all other cases, $Z_{\mathbf{A}}(G)$ is #P-complete. Bulatov and Grohe [2] then gave a substantial generalization of this theorem to all non-negative symmetric matrices $\mathbf{A}$. It basically states that $Z_{\mathbf{A}}(G)$ is computable in P if each *block* of $\mathbf{A}$ has rank at most one, and is #P-hard otherwise. More precisely, decompose $\mathbf{A}$ as a direct sum of $\mathbf{A}_i$ which correspond to the connected components H_i of the undirected graph H defined by the nonzero entries of $\mathbf{A}$. Then, $Z_{\mathbf{A}}(G)$ is computable in P if every $Z_{\mathbf{A}_i}(G)$ is, and #P-hard otherwise. For each non-bipartite H_i, the corresponding $Z_{\mathbf{A}_i}(G)$ is computable in P if $\mathbf{A}_i$ has rank at most one, and is #P-hard otherwise. For each bipartite H_i, the corresponding $Z_{\mathbf{A}_i}(G)$ is computable in P if $\mathbf{A}_i$ has the following form:

$$\mathbf{A}_i = \begin{pmatrix} \mathbf{0} & \mathbf{B}_i \\ {\mathbf{B}_i}^T & 0 \end{pmatrix},$$

where $\mathbf{B}_i$ has rank one, and is #P-hard otherwise. (For considerations of models of computation, strictly speaking the entries of the matrix $\mathbf{A}$ should be algebraic numbers. The same remarks apply below.)

The result of Bulatov and Grohe is both sweeping and applicable. It completely solves the problem for all non-negative symmetric matrices. However, when we are dealing with non-negative matrices, there are no cancellations in the exponential sum $Z_{\mathbf{A}}(G)$. These potential cancellations, when $\mathbf{A}$ is either a real or a complex matrix, may in fact be the source of surprisingly efficient algorithms for computing $Z_{\mathbf{A}}(G)$. The occurrence of these cancellations, or the mere possibility of such occurrence, makes proving any complexity dichotomies more difficult. Such a proof must identify all polynomial-time algorithms utilizing the potential cancellations, such as those found in holographic algorithms [22,21,7], and at the same time carves out exactly what is left. This situation is similar to *monotone* versus *non-monotone* circuit complexity. It turns out that indeed there are more interesting tractable cases over the reals, and in particular, the following 2×2 Hadamard matrix $\mathbf{H}$

$$\mathbf{H} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \tag{1}$$

turns out to be one of such cases.

This is the starting point of the next great chapter on the complexity of $Z_{\mathbf{A}}(\cdot)$. In a beautiful paper [14], Goldberg, Jerrum, Grohe, and Thurley proved a complexity dichotomy theorem for all real-valued symmetric matrices $\mathbf{A}$. The readers should read the paper to see the precise statement, but the main result is a complexity dichotomy criterion on $\mathbf{A}$ such that the problem of computing $Z_{\mathbf{A}}(G)$ for any real $\mathbf{A}$ is either in P or #P-hard. The Hadamard matrix $\mathbf{H}$ and its tensor products $\mathbf{H} \otimes \mathbf{H} \otimes \cdots \otimes \mathbf{H}$ play a major role in the tractable case.

The final dichotomy theorem for $Z_{\mathbf{A}}(G)$ was given in [5]. This theorem applies to all complex-valued symmetric matrices $\mathbf{A}$. It is a long and difficult paper with many components. The final result is that there is a dichotomy criterion on the matrix $\mathbf{A}$. If $\mathbf{A}$ satisfies this criterion then $Z_{\mathbf{A}}(G)$ is computable in polynomial time; if it does not satisfy this criterion then $Z_{\mathbf{A}}(G)$ is #P-hard. The precise statement of the criterion will take many careful steps to formulate, but roughly speaking, the theorem states that in order to be computable in polynomial time, the matrix $\mathbf{A}$ must be a tensor product of Fourier matrices, or a specific kind of modification of such a tensor product by a rank-one matrix. The final tractability uses the exact summability of quadratic Gauss sums. However, even though the dichotomy criterion is intricate to state, it is decidable. This means that given a matrix $\mathbf{A}$, one can decide, in fact decide in polynomial time, whether $\mathbf{A}$ satisfies this tractability criterion.

Next we consider directed graph homomorphism.
For directed graph homomorphism, the matrix $\mathbf{A}$ is in general not symmetric. In a paper that won the best paper award at ICALP in 2006, Dyer, Goldberg, and Paterson [11] proved a dichotomy theorem for $Z_{\mathbf{A}}(\cdot)$, where $\mathbf{A}$ is a 0-1 matrix representing a directed *acyclic* graph H. They introduced the concept of *Lovász-goodness* and proved that $Z_{\mathbf{A}}(\cdot)$ is computable in polynomial time if the graph H is *layered* and *Lovász-good*, and is #P-hard otherwise. A directed acyclic graph is *layered* if one can partition its vertices into k sets $V_1, \ldots, V_k$, for some $k \geq 1$, such that every edge goes from V_i to V_{i+1} for some $i : 1 \leq i < k$. The property of Lovász-goodness turns out to be polynomial-time decidable.

In [3], a dichotomy theorem for the family of all non-negative real matrices $\mathbf{A}$ was proved. We showed that for every fixed non-negative matrix $\mathbf{A}$, the problem of computing $Z_{\mathbf{A}}(\cdot)$ is either in polynomial time or #P-hard. Moreover, the dichotomy criterion is *decidable*: There is a finite-time algorithm which, given any non-negative matrix $\mathbf{A}$, decides whether $Z_{\mathbf{A}}(\cdot)$ is in P or #P-hard. But the complexity of this decision algorithm for the dichotomy criterion is not known to be in polynomial time (in the size of the matrix $\mathbf{A}$.)

3 Counting Constraint Satisfaction Problems

A broader class of locally constrained counting problems is known as the counting Constraint Satisfaction Problems (CSP). For unweighted counting CSP, denoted as #CSP, the problem is defined as follows: Let D be an arbitrary fixed finite domain and let Γ be an arbitrary fixed finite set of constraint relations $\{R_1, \ldots, R_k\}$ over D, where each R_i has some arity r_i. An instance of #CSP(Γ) consists of two parts. The first part is a finite set of variables $X = \{x_1, \ldots, x_n\}$ each taking values in D. The second part is a finite sequence of relations from Γ, each applied to some variables in X. These are the constraints to be satisfied for this instance. Let R be the n-ary relation over D defined to be the Boolean conjunction of these constraints. The (unweighted) #CSP(Γ) problem asks for the cardinality of R, namely the number of assignments satisfying all the constraints in the input instance.

More generally, in a (complex-weighted) #CSP, Γ is replaced by a fixed finite set of constraint functions $\mathcal{F} = \{f_1, \ldots, f_k\}$, where each f_i maps D^{r_i} to the complex numbers $\mathbb{C}$. An instance of #CSP($\mathcal{F}$) consists of variables X, ranging over D, and a sequence of constraint functions from $\mathcal{F}$, each applied to some variables in X. It defines an n-ary function F: For any $(x_1, \ldots, x_n) \in D^n$, $F(x_1, \ldots, x_n)$ is the product of the constraint function evaluations. The output of #CSP($\mathcal{F}$) is the sum of F over all assignments, known as the *partition function* for #CSP($\mathcal{F}$). Clearly unweighted #CSP is the special case where each constraint function f_i is 0-1 valued.

#CSP can encompass an enormous varieties of counting problems which can be expressed by choosing a particular D, and a particular Γ or $\mathcal{F}$. Graph Homomorphism corresponds to the special case of #CSP with a *single binary function* expressed as the matrix $\mathbf{A}$. In 2008, Bulatov [1] gave a dichotomy theorem for all (unweighted) #CSP(Γ). The theorem states that for every D and Γ, #CSP(Γ) is either computable in polynomial time or #P-complete. This proof uses deep structural theorems from Universal Algebra. Dyer and Richerby [13] gave an alternative proof of Bulatov's dichotomy theorem for unweighted #CSP. They also showed that the decision problem, given D and Γ whether #CSP(Γ) satisfies the criterion for being #P-complete, is decidable in NP.

Very recently, this has been extended to the full generality of complex-valued #CSP [4]. Again, one has to deal with the difficult situation when the constraint functions can have cancellations in the partition function since they can take negative or complex values. The proof ideas of graph homomorphisms [5] are used. The only tool from Universal Algebra that is still used is the notion of a Maltsev polymorphism. Another important ingredient of the proof is a data structure of Dyer and Richerby called a frame, which is a succinct representation for a rectangular relation.

While the full complex-valued #CSP dichotomy theorem requires some background to state properly, the dichotomy over the Boolean domain $D = \{0,1\}$ has an independent proof obtained earlier [8], and has a crisp and easily decidable tractability criterion.

Let $F(X)$ be a function from $\{0,1\}^k$ to $\mathbb{C}$. We define the underlying relation, or support, of F by $R_F = \{X \in \{0,1\}^k \mid F(X) \neq 0\}$. We say a relation $R \subseteq \{0,1\}^k$ is affine if it is an affine subspace over $\mathbb{Z}_2$ (including possibly empty). It is composed of solutions to a system of (affine) linear equations over $\mathbb{Z}_2$. Equivalently, it satisfies the property that if $\alpha, \beta, \gamma \in R$, then the bit-wise XOR string $\alpha \oplus \beta \oplus \gamma \in R$. If R_F is affine, we say F has affine support. We also view relations as functions from $\{0,1\}^k$ to $\{0,1\}$.

We define two classes of functions, for which the #CSP problems over the Boolean domain $D = \{0,1\}$ are tractable. Let $\hat{X}$ denote the $k+1$ dimensional column vector $(x_1, x_2, \ldots, x_k, 1)^{\mathrm{T}}$ over the field $\mathbb{Z}_2$. Suppose A is a matrix over $\mathbb{Z}_2$. The characteristic function $\chi_{A\hat{X}}$ denotes the affine relation on inputs $x_1, x_2, \ldots, x_k$, whose value is 1 if $A\hat{X}$ is the zero vector, and 0 otherwise.

We denote by $\mathscr{A}$, the set of all functions *of affine type*, to have the form $\lambda \cdot \chi_{A\hat{X}} \cdot i^{L_1(X)+L_2(X)+\cdots+L_n(X)}$, where $\lambda \in \mathbb{C}$, $i = \sqrt{-1}$, each L_j is a 0-1 indicator function of the form $\langle \alpha_j, \hat{X} \rangle$, where α_j is a $k+1$ dimensional vector, and the dot product $\langle \cdot, \cdot \rangle$ is computed over $\mathbb{Z}_2$. We may compute the dot product as an ordinary integer dot product, and then take the value mod 2, producing an integer value 0 or 1. The additions among $L_j(X)$ are the usual addition in $\mathbb{Z}$. It can be computed mod 4, but not mod 2.

The second class we define is $\mathscr{P}$, the set of all functions of *product type*. These are functions expressible as a product of unary functions (i.e., functions on one variable), binary equality functions and binary disequality functions, on not necessarily disjoint pairs of variables.

Theorem 1. *Suppose $\mathcal{F}$ is a class of functions mapping Boolean inputs to complex numbers. If $\mathcal{F} \subseteq \mathscr{A}$ or $\mathscr{F} \subseteq \mathscr{P}$, then #CSP($\mathcal{F}$) is computable in polynomial time. Otherwise, #CSP($\mathcal{F}$) is #*P*-hard.*

The expressibility as $\lambda i^{L_1(X)+L_2(X)+\cdots+L_n(X)}$ is equivalent to an expression of the form $\lambda' i^{Q(X)}$ where Q is a homogeneous quadratic polynomial over $\mathbb{Z}$ with the additional requirement that every cross term $x_s x_t$ has an even coefficient, where $s \neq t$.

Let us denote a symmetric function F on k Boolean variables by $[f_0, f_1, \ldots, f_k]$, where f_j is the value of F on inputs of weight j. We define

$$\mathscr{F}_1 = \{\lambda([1,0]^{\otimes k} + i^r[0, \quad 1]^{\otimes k}) \mid \lambda \in \mathbb{C}, k = 1, 2, \ldots, \text{ and } r = 0,1,2,3\};$$
$$\mathscr{F}_2 = \{\lambda([1,1]^{\otimes k} + i^r[1,-1]^{\otimes k}) \mid \lambda \in \mathbb{C}, k = 1, 2, \ldots, \text{ and } r = 0,1,2,3\};$$
$$\mathscr{F}_3 = \{\lambda([1, \ i]^{\otimes k} + i^r[1, \ -i]^{\otimes k}) \mid \lambda \in \mathbb{C}, k = 1, 2, \ldots, \text{ and } r = 0,1,2,3\}.$$

These functions are defined above by listing its values as a vector of length 2^k, as in a truth table. They can be explicitly listed in the symmetric function notation as follows, up to an arbitrary constant multiple from $\mathbb{C}$.

1. $[1, 0, 0, \ldots, 0, \pm 1]$;
2. $[1, 0, 0, \ldots, 0, \pm i]$;
3. $[1, 0, 1, 0, \ldots, 0/1]$;
4. $[0, 1, 0, 1, \ldots, 0/1]$;
5. $[1, i, 1, i, \ldots, i/1]$;
6. $[1, -i, 1, -i, \ldots, (-i)/1]$;
7. $[1, 0, -1, 0, 1, 0, -1, 0, \ldots, 0/1/(-1)]$;
8. $[1, 1, -1, -1, 1, 1, -1, -1, \ldots, 1/(-1)]$;
9. $[0, 1, 0, -1, 0, 1, 0, -1, \ldots, 0/1/(-1)]$;
10. $[1, -1, -1, 1, 1, -1, -1, 1, \ldots, 1/(-1)]$.

A function of arity k is called degenerate if it is the tensor product of k unary functions on k variables.

$$\mathscr{D} = \{F \mid F = [a_1, b_1] \otimes [a_2, b_2] \otimes \cdots \otimes [a_k, b_k], \text{ for some } a_i, b_i \in \mathbb{C}\}$$

A binary function is in $\mathscr{D}$ iff its corresponding matrix is singular.

The class $\mathscr{F}_1 \cup \mathscr{F}_2 \cup \mathscr{F}_3$ is the restriction of $\mathscr{A}$ to symmetric functions. More precisely, $\mathscr{F}_1 \cup \mathscr{F}_2 \cup \mathscr{F}_3$ consists of scalar multiples of all unary or non-degenerate symmetric functions in $\mathscr{A}$.

The special case where $r = 1$, $k = 2$ and $\lambda = (1+i)^{-1}$ in $\mathscr{F}_3$ is noteworthy. In this case we get a real valued function $H(0,0) = H(0,1) = H(1,0) = 1$ and $H(1,1) = -1$. The matrix form of this function is the Hadamard matrix $H = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$. If we take $r = 0$, any k and $\lambda = 1$ in $\mathscr{F}_1$ we get the Equality function on k bits. In this way, from the tractability of the family $\mathscr{F}_1 \cup \mathscr{F}_2 \cup \mathscr{F}_3$ we recover the tractability of the graph homomorphism function $Z_H(G)$.

4 Holant Problems

Motivated by holographic algorithms, another framework to capture locally constrained counting problems was proposed [8]. The Holant framework is as follows. D is a finite set called a domain, and $\mathcal{F}$ is a finite set of functions over D. A *signature grid* $\Omega = (G, \mathcal{F}, \pi)$ consists of a labeled graph $G = (V, E)$ where π labels each vertex $v \in V$ with a function $f_v \in \mathcal{F}$. We consider all edge assignments $\xi : E \to D$; f_v takes inputs from its incident edges $E(v)$ at v and outputs values in $\mathbb{C}$. The counting problem on the instance Ω is to compute

$$\text{Holant}_\Omega = \sum_{\xi : E \to D} \prod_{v \in V} f_v(\xi \mid_{E(v)}). \tag{2}$$

For example, if we take $D = \{0, 1\}$ and attach the Exact-One function at every vertex $v \in V$, then Holant_Ω computes exactly the number of perfect matchings of G.

#CSP is the special case of Holant problems where Equality functions of all arities are assumed to be present in $\mathcal{F}$. In fact each #CSP instance can be realized as a signature grid on a bipartite graph where each vertex on the LHS is labeled by a variable x_i attached with an Equality function and each vertex on the RHS is labeled by a constraint function of the #CSP instance. For Holant problems there is a strong interaction with holographic algorithms [21,7], and the issue of cancellation is of the foremost concern. Dichotomy theorems have been achieved in this framework, mostly on domain size 2 (Boolean domain) [8,9]. The primary challenge is to deal with domain size greater than 2. Some progress has been made in domain size 3 [10]. Below we will focus on the Boolean domain.

A Holant problem is parametrized by a set of signatures.

Definition 2. *Given a set of signatures $\mathcal{F}$, we define the counting problem* Holant($\mathcal{F}$) *as:*
Input: A signature grid $\Omega = (G, \mathcal{F}, \pi)$*;*
Output: Holant_Ω*.*

We say a signature set $\mathcal{F}$ is tractable (resp. #P-hard) if the corresponding counting problem Holant($\mathcal{F}$) is tractable (resp. #P-hard). Similarly for a signature f, we say f is tractable (resp. #P-hard) if $\{f\}$ is.

To introduce the idea of holographic reductions, it is convenient to consider bipartite graphs. For a general graph, we can always transform it into a bipartite graph while preserving the Holant value, as follows. For each edge in the graph, we replace it by a path of length 2. (This operation is called the *2-stretch* of the graph and yields the edge-vertex incident graph.) Each new vertex is assigned the binary Equality signature $(=_2) = [1, 0, 1]$. We use Holant $(\mathcal{R} \mid \mathcal{G})$ to denote the Holant problem on bipartite graphs $H = (U, V, E)$, where each signature for a vertex in U or V is from $\mathcal{R}$ or $\mathcal{G}$, respectively. An input instance for this bipartite Holant problem is a bipartite signature grid and is denoted by $\Omega = (H;\ \mathcal{R} \mid \mathcal{G};\ \pi)$. Signatures in $\mathcal{R}$ are considered as row vectors (or covariant tensors); signatures in $\mathcal{G}$ are considered as column vectors (or contravariant tensors).

For a 2-by-2 matrix T and a signature set $\mathcal{F}$, define $T\mathcal{F} = \{g \mid \exists f \in \mathcal{F}$ of arity $n,\ g = T^{\otimes n} f\}$, similarly for $\mathcal{F}T$. Whenever we write $T^{\otimes n} f$ or $T\mathcal{F}$, we view the signatures as column vectors; similarly for $fT^{\otimes n}$ or $\mathcal{F}T$ as row vectors.

Let T be an invertible 2-by-2 matrix. The holographic transformation by T is the following operation: given a signature grid $\Omega = (H;\ \mathcal{R} \mid \mathcal{G};\ \pi)$, for the same graph H, we get a new grid $\Omega' = (H;\ \mathcal{R}T \mid T^{-1}\mathcal{G};\ \pi')$ by replacing each signature in $\mathcal{R}$ or $\mathcal{G}$ with the corresponding signature in $\mathcal{R}T$ or $T^{-1}\mathcal{G}$.

Theorem 3 (Valiant's Holant Theorem [21]). *If there is a holographic transformation mapping signature grid Ω to Ω', then* $\text{Holant}_\Omega = \text{Holant}_{\Omega'}$*.*

Holographic transformations can reveal the inner relationship and equivalence of two apparently different problems.

Consider the following constraint function $f : \{0, 1\}^4 \to \mathbb{C}$. Let the input (x_1, x_2, x_3, x_4) have Hamming weight w, then $f(x_1, x_2, x_3, x_4) = 3, 0, 1, 0, 3$, if $w = 0, 1, 2, 3, 4$, respectively. In symmetric function notation this function is

$f = [3,0,1,0,3]$. What is the counting problem defined by the Holant sum in equation (2) on 4-regular graphs G when $\mathcal{F} = \{f\}$? By definition, this is a sum over all 0-1 edge assignments of products of local evaluations. We only sum over assignments which assign an even number of 1's to the incident edges of each vertex, since $f = 0$ for $w = 1$ and 3. Then each vertex contributes a factor 3 if the 4 incident edges are assigned all 0 or all 1, and contributes a factor 1 if exactly two incident edges are assigned 1. At first sight this problem may look artificial. Let's consider a holographic transformation. Consider the edge-vertex incident graph $H = (E(G), V(G), \{(e,v) \mid v \text{ is incident to } e \text{ in } G\})$ of G. This Holant problem can be expressed in the bipartite form $\text{Holant}\,(=_2 \mid f)$ on H, where $=_2$ is the binary Equality function. Thus, every $e \in E(G)$ is assigned $=_2$, and every $v \in V(G)$ is assigned f. We can write $=_2$ by its truth table $(1,0,0,1)$ indexed by $\{0,1\}^2$. If we apply the holographic transformation $Z = \frac{1}{\sqrt{2}}\left[\begin{smallmatrix} 1 & 1 \\ i & -i \end{smallmatrix}\right]$, then Valiant's Holant Theorem tells us that $\text{Holant}\,(=_2 \mid f)$ is exactly the same as $\text{Holant}\big((=_2)Z^{\otimes 2} \mid (Z^{-1})^{\otimes 4} f\big)$. Here $(=_2)Z^{\otimes 2}$ is a row vector indexed by $\{0,1\}^2$ denoting the transformed function under Z from $(=_2) = (1,0,0,1)$, and $(Z^{-1})^{\otimes 4} f$ is the column vector indexed by $\{0,1\}^4$ denoting the transformed function under Z^{-1} from f. Let $\hat{f}$ be the Exact-Two function on $\{0,1\}^4$. We can write its truth table as a column vector indexed by $\{0,1\}^4$, which has a value 1 at Hamming weight two and 0 elsewhere. In symmetric signature notation, $\hat{f} = [0,0,1,0,0]$. Then we have

$$\begin{aligned} Z^{\otimes 4}\hat{f} &= Z^{\otimes 4}\{\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right] + \cdots + \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]\} \\ &= \tfrac{1}{4}\{\left[\begin{smallmatrix}1\\i\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\i\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\-i\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\-i\end{smallmatrix}\right] + \cdots + \left[\begin{smallmatrix}1\\-i\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\-i\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\i\end{smallmatrix}\right] \otimes \left[\begin{smallmatrix}1\\i\end{smallmatrix}\right]\} \\ &= \tfrac{1}{2}[3,0,1,0,3] \\ &= \tfrac{1}{2}f; \end{aligned}$$

hence $(Z^{-1})^{\otimes 4} f = 2\hat{f}$. (Here we use the elementary fact that $(A \otimes B)(u \otimes v) = Au \otimes Bv$ for tensor products of matrices and vectors.) Meanwhile, Z transforms $=_2$ to the binary Disequality function $\neq_2$:

$$\begin{aligned} (=_2)Z^{\otimes 2} &= \left(\begin{smallmatrix}1 & 0 & 0 & 1\end{smallmatrix}\right) Z^{\otimes 2} \\ &= \left\{\left(\begin{smallmatrix}1 & 0\end{smallmatrix}\right)^{\otimes 2} + \left(\begin{smallmatrix}0 & 1\end{smallmatrix}\right)^{\otimes 2}\right\} Z^{\otimes 2} \\ &= \tfrac{1}{2}\left\{\left(\begin{smallmatrix}1 & 1\end{smallmatrix}\right)^{\otimes 2} + \left(\begin{smallmatrix}i & -i\end{smallmatrix}\right)^{\otimes 2}\right\} \\ &= [0,1,0] \\ &= (\neq_2). \end{aligned}$$

Hence, up to a global constant factor of 2^n on a graph with n vertices, the Holant problem with $[3,0,1,0,3]$ is exactly the same as $\text{Holant}\,(\neq_2 \mid [0,0,1,0,0])$. A moment's reflection shows that this latter problem is counting the number of Eulerian orientations on 4-regular graphs, a well-studied problem. Thus holographic transformations can reveal the fact that these two problems are really the same problem.

A signature f (resp. a signature set $\mathcal{F}$) is $\mathscr{A}$-transformable if there exists a holographic transformation T such that $f \in T\mathscr{A}$ (resp. $\mathcal{F} \subseteq T\mathscr{A}$) and $[1,0,1]T^{\otimes 2} \in \mathscr{A}$, where $\mathscr{A}$ is the affine family. Similarly, a signature f (resp. a signature set $\mathcal{F}$) is $\mathscr{P}$-transformable if there exists a holographic transformation T such that $f \in T\mathscr{P}$ (resp. $\mathcal{F} \subseteq T\mathscr{P}$) and $[1,0,1]T^{\otimes 2} \in \mathscr{P}$, where $\mathscr{P}$ is the product type family. These two families are tractable because after a transformation by T, it is a tractable #CSP instance.

Definition 4. *A set of signatures $\mathcal{F}$ is called* vanishing *if the value* Holant_Ω *is zero for every signature grid Ω using the functions in $\mathcal{F}$. A signature f is called* vanishing *if the singleton set $\{f\}$ is vanishing.*

Definition 5. *An arity n symmetric signature of the form $f = [f_0, f_1, \ldots, f_n]$ is in $\mathscr{R}_t^+$ for a nonnegative integer $t \geq 0$ if $t > n$ or for any $0 \leq k \leq n-t$, $f_k, \ldots, f_{k+t}$ satisfy the recurrence relation*

$$\binom{t}{t} i^t f_{k+t} + \binom{t}{t-1} i^{t-1} f_{k+t-1} + \cdots + \binom{t}{0} i^0 f_k = 0. \tag{3}$$

We define $\mathscr{R}_t^-$ similarly but with $-i$ in place of i in (3).

It is easy to see that $\mathscr{R}_0^+ = \mathscr{R}_0^-$ is the set of all zero signatures. Also, for $\sigma \in \{+,-\}$, we have $\mathscr{R}_t^\sigma \subseteq \mathscr{R}_{t'}^\sigma$ when $t \leq t'$. By definition, if $\text{arity}(f) = n$ then $f \in \mathscr{R}_{n+1}^\sigma$.

Let $f = [f_0, f_1, \ldots, f_n] \in \mathscr{R}_t^+$ with $0 < t \leq n$. Then the characteristic polynomial of its recurrence relation is $(1+xi)^t$. Thus there exists a polynomial $p(x)$ of degree at most $t-1$ such that $f_k = i^k p(k)$, for $0 \leq k \leq n$. This statement extends to $\mathscr{R}_{n+1}^+$ since a polynomial of degree n can interpolate any set of $n+1$ values. Furthermore, such an expression is unique. If there are two polynomials $p(x)$ and $q(x)$, both of degree at most n, such that $f_k = i^k p(k) = i^k q(k)$ for $0 \leq k \leq n$, then $p(x)$ and $q(x)$ must be the same polynomial. Now suppose $f_k = i^k p(k)$ $(0 \leq k \leq n)$ for some polynomial p of degree at most $t-1$, where $0 < t \leq n$. Then f satisfies the recurrence (3) of order t. Hence $f \in \mathscr{R}_t^+$.

Thus $f \in \mathscr{R}_{t+1}^+$ iff there exists a polynomial $p(x)$ of degree at most t such that $f_k = i^k p(k)$ $(0 \leq k \leq n)$, for all $0 \leq t \leq n$. For $\mathscr{R}_{t+1}^-$, just replace i by $-i$.

Definition 6. *For a nonzero symmetric signature f of arity n, it is of* positive *(resp.* negative*)* recurrence degree *$t \leq n$, denoted by* $\text{rd}^+(f) = t$ *(resp.* $\text{rd}^-(f) = t$*), if and only if $f \in \mathscr{R}_{t+1}^+ - \mathscr{R}_t^+$ (resp. $f \in \mathscr{R}_{t+1}^- - \mathscr{R}_t^-$). If f is the all zero signature, we define* $\text{rd}^+(f) = \text{rd}^-(f) = -1$.

Lemma 7. *Let $f = [f_0, \ldots, f_n]$ be a symmetric signature of arity n, not identically 0. Then for any nonnegative integer $0 \leq t < n$ and $\sigma \in \{+,-\}$, the following are equivalent:*

(i) There exist t unary signatures $v_1, \ldots, v_t$, such that

$$f = \text{Sym}_n^{n-t}([1, \sigma i]; v_1, \ldots, v_t). \tag{4}$$

where $\text{Sym}_n^{n-t}([1,\sigma i]; v_1, \ldots, v_t)$ is the symmetrized version of the tensor product of $n-t$ copies of $[1, \sigma i]$ and $v_1, \ldots, v_t$.

(ii) $f \in \mathscr{R}^{\sigma}_{t+1}$.

In terms of rd^{σ} we can prove

$$\mathscr{V}^{\sigma} = \{f \mid 2\,\mathrm{rd}^{\sigma}(f) < \mathrm{arity}(f)\}.$$

The latest Holant problem dichotomy concerns an arbitrary set of symmetric complex-valued local constraint functions on the Boolean domain [6].

Theorem 8. *Let $\mathcal{F}$ be any set of symmetric, complex-valued signatures in Boolean variables. Then* Holant$(\mathcal{F})$ *is #P-hard unless $\mathcal{F}$ satisfies one of the following conditions, in which case the problem is in P:*

1. *All non-degenerate signatures in $\mathcal{F}$ are of arity at most 2;*
2. *$\mathcal{F}$ is $\mathscr{A}$-transformable;*
3. *$\mathcal{F}$ is $\mathscr{P}$-transformable;*
4. *$\mathcal{F} \subseteq \mathscr{V}^{\sigma} \cup \{f \in \mathscr{R}^{\sigma}_{2} \mid \mathrm{arity}(f) = 2\}$ for $\sigma \in \{+, -\}$;*
5. *All non-degenerate signatures in $\mathcal{F}$ are in $\mathscr{R}^{\sigma}_{2}$ for $\sigma \in \{+, -\}$.*

Valiant introduced a novel class of algorithms called holographic algorithms using matchgates [21,22,7]. In terms of the type of problems that can be solved by holographic algorithms using matchgates, we have gained a substantial knowledge, both about what they can do and what they can't do. There are some very strong dichotomy theorems that pertain to matchgates based holographic algorithms. They are mainly still restricted to symmetric function sets $\mathcal{F}$ on Boolean variables. They generally indicate that the complexity of the counting problems defined by $\mathcal{F}$ (in both the #CSP and in Holant frameworks) can be classified into exactly three classes, depending on $\mathcal{F}$: Those that can be solved in polynomial time in general; those that are #P-hard in general but solvable in polynomial time over planar structures; and those that remain #P-hard over planar structures. Furthermore, the second class is precisely those which can be realized by matchgates under a holographic transformation, and then the polynomial time algorithm over the planar instances is the Fisher-Kasteleyn-Temperley algorithm under a holographic transformation. To prove this in full generality is an outstanding open problem. See [9,15] for details.

References

1. Bulatov, A.A.: The Complexity of the Counting Constraint Satisfaction Problem. In: Aceto, L., Damgard, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 646–661. Springer, Heidelberg (2008)
2. Bulatov, A., Grohe, M.: The complexity of partition functions. Theoretical Computer Science 348(2), 148–186 (2005)
3. Cai, J.-Y., Chen, X.: A decidable dichotomy theorem on directed graph homomorphisms with non-negative weights. In: Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science, pp. 437–446 (2010)

4. Cai, J.-Y., Chen, X.: Complexity of counting CSP with complex weights. In: Proceedings of the 44th Symposium on Theory of Computing (STOC 2012), pp. 909–920 (2012), `http://arxiv.org/abs/1111.2384`
5. Cai, J.-Y., Chen, X., Lu, P.: Graph Homomorphisms with Complex Values: A Dichotomy Theorem. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part I. LNCS, vol. 6198, pp. 275–286. Springer, Heidelberg (2010), `http://arxiv.org/abs/0903.4728`
6. Cai, J.-Y., Guo, H., Williams, T.: A Complete Dichotomy Rises from the Capture of Vanishing Signatures, `http://arxiv.org/abs/1204.6445`
7. Cai, J.-Y., Lu, P.: Holographic algorithms: from art to science. In: STOC 2007: Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing, pp. 401–410. ACM, New York (2007); Journal of Computer and System Sciences 77(1), 41–61 (2011)
8. Cai, J.-Y., Lu, P., Xia, M.: Holant problems and counting CSP. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, pp. 715–724 (2009)
9. Cai, J.-Y., Lu, P., Xia, M.: Holographic algorithms with matchgates capture precisely tractable planar #CSP. In: Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science, pp. 427–436 (2010)
10. Cai, J.-Y., Lu, P., Xia, M.: Dichotomy for Holant* Problems with a Function on Domain Size 3. To appear in SODA 2013 (2013), Full paper available at `http://arxiv.org/abs/1207.2354`
11. Dyer, M., Goldberg, L.A., Paterson, M.: On Counting Homomorphisms to Directed Acyclic Graphs. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006, Part I. LNCS, vol. 4051, pp. 38–49. Springer, Heidelberg (2006); Journal of the ACM (JACM) 54(6), Article No. 27 (December 2007)
12. Dyer, M.E., Greenhill, C.: The complexity of counting graph homomorphisms. Random Structures & Algorithms 17(3-4), 260–289 (2000)
13. Dyer, M.E., Richerby, D.M.: On the Complexity of #CSP. In: Proceedings of the 42nd ACM Symposium on Theory of Computing, pp. 725–734 (2010)
14. Goldberg, L.A., Grohe, M., Jerrum, M., Thurley, M.: A complexity dichotomy for partition functions with mixed signs. SIAM Journal on Computing 39(7), 3336–3402 (2010)
15. Guo, H., Williams, T.: The Complexity of Planar Boolean #CSP with Complex Weights, `http://arxiv.org/abs/1212.2284`
16. Hell, P., Nešetřil, J.: Graphs and Homomorphisms. Oxford University Press (2004)
17. Lovász, L.: Operations with structures. Acta Mathematica Hungarica 18, 321–328 (1967)
18. Schaefer, T.J.: The complexity of satisfiability problems. In: Proceedings of the 10th Annual ACM Symposium on Theory of Computing, pp. 216–226 (1978)
19. Toda, S.: PP is as Hard as the Polynomial-Time Hierarchy. SIAM Journal on Computing 20, 865–877 (1991)
20. Valiant, L.G.: The Complexity of Computing the Permanent. Theoretical Computer Science (Elsevier) 8(2), 189–201
21. Valiant, L.G.: Holographic algorithms. SIAM J. Comput. 37(5), 1565–1594 (2008)
22. Valiant, L.G.: Accidental algorithms. In: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 509–517 (2006)

Algorithms for Analyzing and Verifying Infinite-State Recursive Probabilistic Systems

Kousha Etessami

School of Informatics, University of Edinburgh, UK
kousha@inf.ed.ac.uk

Until recent years, research on automated verification and model checking of probabilistic systems was largely confined to using *finite-state* Markov chains (MCs) and *finite-state* Markov Decision processes (MDPs) as the core underlying models. In recent years, researchers have begun to develop new algorithms for, and study the computational complexity of, analysis and verification problems for classes of finitely-presented infinite-state probabilistic systems that arise as probabilistic extensions to classic infinite-state automata-theoretic models.

A number of important countable infinite-state stochastic processes (including multi-type branching processes, stochastic context-free grammars, and quasi-birth-death processes), which are all closely related to classic automata-theoretic and process-algebraic models, can be suitably captured by subclasses of *recursive Markov chains* and *recursive Markov decision processes*, which are obtained by adding a natural recursion feature to finite-state MCs and MDPs. Recursive MCs and MDPs provide natural abstract models of probabilistic procedural programs with recursion, and they are expressively equivalent to probabilistic and MDP extensions of pushdown automata.

Key computational problems for analyzing recursive MCs, and for analyzing important subclasses of resursive MDPs and recursive stochastic games, can all be boiled down to computing the *least fixed point* (LFP) solution of corresponding *monotone* systems of nonlinear polynomial (min/max) equations.

In this talk I will survey algorithms for, and discuss the complexity of, some key analysis and model checking problems for (sub)classes of *recursive* MCs, MDPs, and stochastic games. In particular, I will discuss recent joint work with Alistair Stewart and Mihalis Yannakakis (in papers that appeared at STOC'12 and ICALP'12), in which we have obtained polynomial time algorithms for computing, to within arbitrary desired precision, the extinction probabilities of multi-type branching processes, the probability that an arbitrary stochastic context-free grammar generates a given string, and the optimum (maximum or minimum) extinction probabilities for branching MDPs. This requires computing the LFP solution of corresponding monotone systems of probabilistic min/max polynomial equations, which amount to Bellman optimality equations for minimizing/maximizing extinction and termination probabilities for branching MDPs, context-free MDPs, and 1-exit recursive MDPs. Our algorithms combine generalizations of Newton's method with other techniques, including linear programming, to compute the LFP within desired precision in P-time. The algorithms are fairly easy to implement, but analyzing their running time is involved.

Recursion Schemes, Collapsible Pushdown Automata and Higher-Order Model Checking

Luke Ong

University of Oxford, UK

Abstract. This paper is about two models of computation that underpin recent developments in the algorithmic verification of higher-order computation. Recursion schemes are in essence the simply-typed lambda calculus with recursion, generated from first-order symbols. Collapsible pushdown automata are a generalisation of pushdown automata to higher-order stacks — which are iterations of stack of stacks — that contain symbols equipped with links. We study and compare the expressive power of the two models and the algorithmic properties of infinite structures such as trees and graphs generated by them. We conclude with a brief overview of recent applications to the model checking of higher-order functional programs. A central theme of the work is the fruitful interplay of ideas between the neighbouring fields of semantics and algorithmic verification.

1 Introduction

Over the past decade, there has been significant progress in the development of finite-state and pushdown model checking for software verification. Though highly effective when applied to first-order imperative programs such as C, these techniques are less useful for higher-order functional programs. In contrast, the standard approaches to the verification of higher-order programs are *type-based static analysis* on the one hand, and *theorem proving* and *dependent types* on the other. The former is sound but imprecise; the latter typically requires human intervention.

Recently an approach to model checking higher-order programs based on *recursion schemes* has emerged as a verification methodology that promises to combine accurate analysis with push-button automation. A grammar for generating possibly-infinite trees, recursion schemes are in essence the simply-typed λ-calculus with recursion, built up from first-order symbols. Ong [53] proved that the trees generated by recursion schemes have decidable monadic second-order (MSO) theories, subsuming earlier well-known MSO decidability results for regular (order-0) [63] and algebraic (order-1) trees [15]. Building on [53], Kobayashi [39] introduced a novel approach to the verification of higher-order functional programs by reduction to the *recursion schemes model checking problem*: does the tree generated by a given recursion schemes satisfy a given correctness property?

This survey paper concerns two models of higher-order computation: recursion schemes and collapsible pushdown automata. Recursive program schemes are an old formalism for the semantical analysis of both imperative and functional programs [52,19,14]. Recursion schemes, which generalise recursive program schemes to higher orders, are a compelling model of higher-order functional programs. Pushdown automata characterise the control flow of first-order recursive programs [34]; pushdown model checkers (such as Moped [26]) are an important component of state-of-the-art software model checkers. The two models are suitable for the algorithmic verification of higher-order computation. On the one hand, they model higher-order computation accurately: recursion schemes are a version of PCF [62]; and collapsible pushdown automata compute exactly the innocent strategies, which give rise to the fully abstract model of PCF [32,49,29]. On the other, they enjoy rich algorithmic properties [36,12,53,29].

The goal of our work is to use semantic methods, in conjunction with algorithmic ideas and techniques from verification, to formally analyse programming situations in which higher-order features are important. In Section 2, we present two families of generators of infinite structures: recursion schemes and higher-order pushdown automata; we study their relationship and characterise their expressivity. In Section 3, we survey recent results on the model checking of trees generated by recursion schemes. In Section 4, we introduce collapsible pushdown automata, study their relationship with recursion schemes, and discuss developments in the solution of parity games over the configuration graphs of collapsible pushdown systems. In Section 5, we briefly discuss recent applications to the model checking of higher-order functional programs, and then conclude.

2 Two Families of Generators of Infinite Structures

2.1 Higher-Order Pushdown Automata

Higher-order pushdown automata were introduced by Maslov [45,46] as a generalisation of pushdown automata and nested pushdown automata. Let Γ be a stack alphabet that contains a distinguished *bottom-of-stack symbol* $\bot$. An *order-0 stack* is just a stack symbol. An *order-*$(n+1)$ *stack* is a non-null sequence (written $[s_1 \cdots s_l]$) of order-n stacks. We often abbreviate order-n stack to n-stack, and write $n\text{-}\mathit{Stack}_\Gamma$ for the set of n-stacks over Γ. As usual, $\bot$ cannot be popped from or pushed onto a stack. (Thus we require an *order-1 stack* to be a non-null sequence $[a_1 \cdots a_l]$ of Γ-symbols such that for all $1 \le i \le l$, $a_i = \bot$ if and only if $i = 1$.) We define $\bot_k$, the *empty k-stack*: $\bot_0 := \bot$ and $\bot_{k+1} := [\bot_k]$. When displaying examples of n-stacks, we shall omit $\bot$ to avoid clutter.

Operations on n-Stacks The following operations are defined on 1-stacks:

$$\begin{array}{rll} \mathit{push}_1^Z\ [Z_1 \cdots Z_{i-1}\, Z_i] & := [Z_1 \cdots Z_{i-1},\, Z_i, Z] & \text{where } Z \in \Gamma \setminus \{\,\bot\,\} \\ \mathit{pop}_1\ [Z_1 \cdots Z_{i-1}\, Z_i] & := [Z_1 \cdots Z_{i-1}] & \text{where } Z_i \neq \bot \\ \mathit{top}_1\ [Z_1 \cdots Z_{i-1}\, Z_i] & := Z_i & \end{array}$$

The following operations are defined on n-stacks, where $n \geq 2$:

$$\begin{aligned}
push_n\,[s_1 \cdots s_{i-1}\, s_i] &:= [s_1 \cdots s_{i-1}\, s_i\, s_i] \\
pop_n\,[s_1 \cdots s_{i-1}\, s_i] &:= [s_1 \cdots s_{i-1}] \\
push_k\,[s_1 \cdots s_{i-1}\, s_i] &:= [s_1 \cdots s_{i-1} push_k\, s_i] \quad \text{where } 2 \leq k < n \\
push_1^Z\,[s_1 \cdots s_{i-1}\, s_i] &:= [s_1 \cdots s_{i-1}\, push_1^Z\, s_i] \quad \text{where } Z \in \mathit{\Gamma} \setminus \{\, \perp \,\} \\
pop_k\,[s_1 \cdots s_{i-1}\, s_i] &:= [s_1 \cdots s_{i-1}\, pop_k\, s_i] \quad \text{where } 1 \leq k < n \\
top_n\,[s_1 \cdots s_{i-1}\, s_i] &:= s_i \\
top_k\,[s_1 \cdots s_{i-1}\, s_i] &:= top_k\, s_i \quad \text{where } 1 \leq k < n
\end{aligned}$$

For $1 \leq k \leq n$, the operation pop_k is undefined on any n-stack such that its top k-stack (or the n-stack itself, in case $k = n$) has only one element. For example $pop_2\,[[\perp \alpha\, \beta]]$ and $pop_1\,[[\perp \alpha\, \beta]\,[\perp]]$ are both undefined. For $n \geq 0$, we define the set Op_n of *order-n stack operations*:

$$\begin{aligned}
Op_1 &:= \{\, push_1^Z \mid Z \in \mathit{\Gamma} \setminus \{\, \perp \,\} \,\} \cup \{\, pop_1 \,\} \\
n \geq 2,\ Op_n &:= \{\, push_k, pop_k \mid 2 \leq k \leq n \,\} \cup Op_1.
\end{aligned}$$

Higher-order pushdown automata were first used to define word languages. Here we take a somewhat generic approach to the definition.

Definition 1. An *abstract store system* is a tuple $\langle \mathit{\Gamma}, AStore_{\mathit{\Gamma}}, Op, top, \perp \rangle$ where $\mathit{\Gamma}$ is a (finite) store alphabet, $AStore_{\mathit{\Gamma}}$ is a set of *abstract stores* notionally generated from $\mathit{\Gamma}$, Op is a set of *abstract store operations* which are just partial functions $AStore_{\mathit{\Gamma}} \rightharpoonup AStore_{\mathit{\Gamma}}$, $top : AStore_{\mathit{\Gamma}} \to \mathit{\Gamma}$ is an *abstract read* function, and $\perp \in AStore_{\mathit{\Gamma}}$ is the initial abstract store.

For example, for $n \geq 0$, the tuple $\langle \mathit{\Gamma}, n\text{-}Stack_{\mathit{\Gamma}}, Op_n, top_1, \perp_n \rangle$, which we shall call the *system of order-n stacks over* $\mathit{\Gamma}$, is an abstract store system. Another example is the *system of order-n collapsible stacks*, which will be introduced in Section 4.1. The semi-infinite tape (of a Turing machine) and the FIFO queue (of a Minsky machine) are also examples of abstract store system.

Definition 2. (i) Let $\mathcal{S} = \langle \mathit{\Gamma}, AStore_{\mathit{\Gamma}}, Op, top, \perp \rangle$ be an abstract store system. A *word-language* $\mathcal{S}$*-automaton* is a tuple $\mathcal{A} = \langle \mathcal{S}, Q, \mathit{\Sigma}, \mathit{\Delta}, q_I, F \rangle$ where Q is a finite set of states, $\mathit{\Sigma}$ is an input alphabet, $\mathit{\Delta} \subseteq Q \times (\mathit{\Sigma} \cup \{\, \epsilon \,\}) \times \mathit{\Gamma} \times Op \times Q$ is a transition relation, $q_I \in Q$ is the initial state, and $F \subseteq Q$ is a set of final states. A *configuration* is a pair (q, s) where $q \in Q$ and $s \in AStore_{\mathit{\Gamma}}$; the initial configuration is $(q_I, \perp)$ where $\perp \in AStore_{\mathit{\Gamma}}$. The transition relation $\mathit{\Delta}$ induces a transition relation between configurations according to the rule: if $(q, a, Z, \theta, q') \in \mathit{\Delta}$ and $top(s) = Z$ then $(q, s) \xrightarrow{a} (q', \theta(s))$. A word $w \in \mathit{\Sigma}^*$ is *accepted* by $\mathcal{A}$ just in case there is a sequence of transitions of the form $(q_I, \perp) \xrightarrow{a_1} (q_1, s_1) \xrightarrow{a_2} \cdots \xrightarrow{a_m} (q_m, s_m)$ such that $q_m \in F$, $s_m = \perp$ and $w = a_1 \cdots a_m$.

(ii) We say that a word-language $\mathcal{S}$-automaton is *deterministic* just if $\mathit{\Delta}$ is a partial function $Q \times (\mathit{\Sigma} \cup \{\, \epsilon \,\}) \times \mathit{\Gamma} \rightharpoonup Op \times Q$; further, for every q and Z, if $\mathit{\Delta}$ is defined on (q, ϵ, Z), then it must be undefined on (q, a, Z) for every $a \in \mathit{\Sigma}$.

(iii) In case $\mathcal{S} = \langle \Gamma, n\text{-}Stack_\Gamma, Op_n, top_1, \bot_n \rangle$ is the system of n-stacks over Γ, we refer to a word-language $\mathcal{S}$-automaton as an *order-n pushdown word-language automaton*, and specify it as $\langle \Gamma, Q, \Sigma, \Delta, q_I, F \rangle$. When it is clear from the context, we call it an order-n pushdown automaton or simply n-PDA.

By definition, order-0 PDA are finite-state automata. Order-1 PDA are (ordinary) PDA. Order-2 PDA capture the *indexed languages* of Aho [4]. There are several major studies on higher-order PDA in the literature [19,25]. [1]

Example 3. (i) The language $L = \{ a^n \, b^n \, c^n \mid n \geq 0 \}$ is recognisable by an order-2 deterministic PDA $\langle \{ \bot, Z \}, \{ q_1, q_2, q_3 \}, \{ a, b, c \}, \delta, q_1, \{ q_1, q_3 \} \rangle$ where $\delta : Q \times (\Sigma \cup \{ \epsilon \}) \times \Gamma \rightharpoonup Op_2^* \times Q$ is defined as follows:

$$\begin{array}{lll} (q_1, a, \bot) \mapsto (push_2 \, ; push_1^Z, q_1) & (q_1, b, Z) \mapsto (pop_1, q_2) & (q_2, c, \bot) \mapsto (pop_2, q_3) \\ (q_1, a, Z) \mapsto (push_2 \, ; push_1^Z, q_1) & (q_2, b, Z) \mapsto (pop_1, q_2) & (q_3, c, Z) \mapsto (pop_2, q_3) \end{array}$$

For example, the following computation accepts $a^2 \, b^2 \, c^2$.

$$\begin{tikzcd}
q_1 \; \texttt{[[]]} \arrow[r, "a"] & q_1 \; \texttt{[[][}Z\texttt{]]} \arrow[r, "a"] & q_1 \; \texttt{[[][}Z\texttt{][}ZZ\texttt{]]} \arrow[d, "b"] \\
 & & q_2 \; \texttt{[[][}Z\texttt{][}Z\texttt{]]} \arrow[d, "b"] \\
q_3 \; \texttt{[[]]} & q_3 \; \texttt{[[][}Z\texttt{]]} \arrow[l, "c"'] & q_2 \; \texttt{[[][}Z\texttt{][]]} \arrow[l, "c"']
\end{tikzcd}$$

(ii) Using the same idea, we can use order-2 deterministic pushdown automata to recognise $\{ a^n \, b^n \, c^n \, d^n \mid n \geq 0 \}$ and so on.

Several basic properties of the pushdown hierarchy of word languages were proved by Maslov [45,46].

Theorem 4 (Maslov 1974). *Let $n \geq 0$.*

(i) The emptiness problem for order-n PDA is decidable.

(ii) The order-n languages form an abstract family of languages[2]*.*

(iii) Higher-order PDA define an infinite hierarchy of word languages.

In the sequel, we shall use higher-order pushdown automata and a new variant, called *collapsible pushdown automata*, to generate infinite structures such as languages of infinite trees (Section 4.1) and infinite graphs (Section 4.2). These structures have rich algorithmic properties.

[1] Engelfriet's work used a somewhat different (but equivalent) machine called *iterated pushdown automata.*

[2] An *abstract family of languages* (AFL) is a collection of languages closed under union, concatenation, Kleene star, intersection with regular languages, homomorphism and inverse homomorphism.

2.2 Recursion Schemes

Recursion schemes are a method of constructing possibly infinite term-trees (or sets of such trees). The idea goes back to Park's pioneering work on program schemes and fixpoint theory [57] in the late 1960s and Patterson's PhD thesis [61]. Program schemes are a program calculus that clearly separates control structure and operations on data, thus providing a framework for investigating purely structural program transformation and the descriptive power of control structures. There is a large literature [5,51,52,28]throughout the 1970s and much of the 1980s on the semantics and transformation of recursive program schemes. In the late 1970s Damm, Fehr and Indermark [18,20,19] introduced program schemes constrained by a family of simple types, and considered them as generators of finite-word and tree languages. For a survey of the early work, see Courcelle's handbook article [14].

For us, *types* are simple types defined by the grammar $A ::= o \mid A \to B$. By convention, arrows associate to the right; thus every type can be written uniquely as $A_1 \to \cdots \to A_n \to o$, for some $n \geq 0$. We define the *order* of a type: $ord(o) := o$ and $ord(A \to B) := \max(ord(A) + 1, ord(B))$. Intuitively the order of a type measures how deeply nested it is on the left of the arrow.

Assume a countably infinite set Var of typed variables. Let Θ be a set of typed symbols such as Var; we write $s : A$ to mean s has type A. The set of *applicative terms* generated from Θ is the least set containing Θ closed under the *application rule*: if $s : A \to B$ and $t : A$ then $s\,t : B$. By convention, application associates to the left. Given a term $s : A$, we define $ord(s) := ord(A)$.

Definition 5. A *higher-order recursion scheme* (or simply *recursion scheme*) is a tuple $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ where

- Σ is a *ranked alphabet of terminals* i.e. each $f \in \Sigma$ has an arity $\mathsf{ar}(f) \geq 0$, which is written $f : \mathsf{ar}(f)$ by abuse of notation; we assume that f has the type $\underbrace{o \to \cdots \to o \to}_{\mathsf{ar}(f)} o$.
- $\mathcal{N}$ is a set of typed *non-terminals*; $S \in \mathcal{N}$ is a distinguished *start symbol* of type o.
- $\mathcal{R}$ is a finite set of rewrite rules of the form $F\,\xi_1 \cdots \xi_n \;\to\; e$ where $F : A_1 \to \cdots \to A_n \to o$, each $\xi_i : A_i$, and $e : o$ is an applicative term generated from $\Sigma \cup \mathcal{N} \cup \{\, \xi_1, \cdots, \xi_n \,\}$.

The *order* of a recursion scheme is defined to be the highest order (of the type) of its non-terminals. In the following we shall use uppercase letters F, G, H, etc. to range over non-terminals, lowercase letters f, g, h, etc. to range over terminals, and $\xi, \varphi, \psi, x, y, z$, etc. to range over variables.

We use *deterministic* recursion schemes (i.e. at most one rewrite rule for each non-terminal) to define possibly-infinite trees. (In general, recursion schemes define tree languages. Henceforth, when defining trees, we assume that recursion schemes are deterministic.) Given a recursion scheme G, the term-tree generated by G, denoted $[\![G]\!]$, is the possibly-infinite applicative term *constructed from the terminals*, which is obtained from the start symbol S by unfolding the rewrite rules *ad infinitum*, replacing formal by actual parameters each time.

Example 6 (An order-1 recursion scheme G_1). Take the ranked alphabet $\Sigma = \{ f : 2, g : 1, a : 0 \}$. Consider the order-1 recursion scheme G_1 with rewrite rules:

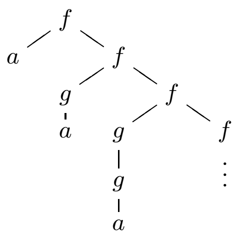
$$S \to F\,a \qquad F\,x \to f\,x\,(F\,(g\,x))$$

where $F : o \to o$. Unfolding from the start symbol S, we have

$$S \to F \to f\,a\,(F\,(g\,a)) \to f\,a\,(f\,(g\,a)\,(F\,(g\,(g\,a)))) \to \cdots$$

thus generating the infinite applicative term

$$f\,a\,(f\,(g\,a)\,(f\,(g\,(g\,a))(\cdots))).$$



Because the rewrite system is Church-Rosser, it does not matter which reduction strategy is used provided it is *fair* in the sense of not neglecting any branch. The tree generated by G_1, $[\![G_1]\!]$, is the abstract syntax tree of the infinite term as shown on the right.

Formally the rewrite relation $\to_G$ is defined by induction over the rules:

$$\frac{F\xi_1 \cdots \xi_n \to e \text{ is a } \mathcal{R}\text{-rule}}{Ft_1 \cdots t_n \to_G e[t_1/\xi_1, \cdots, t_n/\xi_n]} \qquad \frac{t \to_G t'}{s\,t \to_G s\,t'} \qquad \frac{t \to_G t'}{t\,s \to_G t'\,s}$$

We write $\to_G^*$ for the reflexive, transitive closure of $\to_G$.

Let $l := \max\{ \mathsf{ar}(f) \mid f \in \Sigma \}$. A *$\Sigma$-labelled tree* is a partial function t from $\{ 1, \cdots, l \}^*$ to Σ such that $dom(t)$ is prefix-closed; we assume that t is *ranked* i.e. if $t(w) = a$ and $\mathsf{ar}(a) = m$ then $\{ i \mid w\,i \in dom(t) \} = \{ 1, \cdots, m \}$. A (possibly infinite) sequence π over $\{ 1, \cdots, l \}$ is a *path* of t if every finite prefix of π is in $dom(t)$. Given a term t, we define a (finite) tree $t^\perp$ by:

$$t^\perp := \begin{cases} f & \text{if } t \text{ is a terminal } f \\ {t_1}^\perp {t_2}^\perp & \text{if } t \text{ is of the form } t_1 t_2 \text{ and } {t_1}^\perp \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

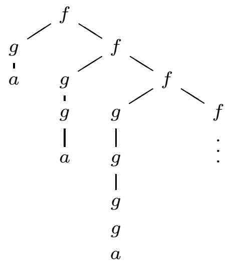
For example $(f\ (F\ a)\ b)^\perp = f\ \perp\ b$. Let $\sqsubseteq$ be the partial order on $\Sigma \cup \{ \perp \}$ defined by $\forall a \in \Sigma . \perp \sqsubseteq a$. We extend $\sqsubseteq$ to a partial order on trees by: $t \sqsubseteq s := \forall w \in dom(t).(w \in dom(s) \wedge t(w) \sqsubseteq s(w))$. For example, $\perp \sqsubseteq f\ \perp\ \perp \sqsubseteq f\ \perp\ b \sqsubseteq f\ a\ b$. For a directed set T of trees, we write $\bigsqcup T$ for the least upper bound of elements of T with respect to $\sqsubseteq$. We define the *tree generated by G*, or the *value tree* of G, by $[\![G]\!] := \bigsqcup\{ t^\perp \mid S \to_G^* t \}$. By construction, $[\![G]\!]$ is a possibly infinite, ranked $(\Sigma \cup \{ \perp \})$-labelled tree. For $n \geq 0$, we write $\mathit{RecSchTree}_n^\Sigma$ for the class of Σ-labelled trees generated by order-n recursion schemes.

Example 7 (An order-2 recursion scheme G_2).

Take $\Sigma = \{ f : 2, g : 1, a : 0 \}$, with rewrite rules:

$$\begin{aligned} S &\to F\, g \\ B\, \varphi\, \psi\, x &\to \varphi\, (\psi\, x) \\ F\, \varphi &\to f\, (\varphi\, a)\, (F\, (B\, \varphi\, \varphi)) \end{aligned}$$

where $B : (o \to o) \to (o \to o) \to o \to o$ and $F : (o \to o) \to o$. The value tree, $[\![G_2]\!] : \{ 1, 2 \}^* \longrightarrow \Sigma$, is shown on the right.



Recursion Schemes as Generators of Word Languages. By viewing a word as a linear tree i.e. a tree with branching factor at most one, we can use (non-deterministic) recursion schemes to generate word languages. Thus a finite word "$a\, b\, c$" (say) is represented by the applicative term $a\, (b\, (c\ e))$, where a, b and c are now regarded as terminal symbols of arity 1, and e is a distinguished nullary end-of-word marker. We call such a recursion scheme a *word-language recursion scheme.*

Example 8. (i) The language $\{ a^n\, b^n \mid n \geq 0 \}$ is generated by the order-1 recursion scheme: $S \to F\, e \quad F\, x \to a\, (F\, (b\, x)) \quad F\, x \to x$

(ii) The language $\{ a^n\, b^n\, c^n \mid n \geq 0 \}$ is generated by the order-2 recursion scheme:

$$\begin{array}{lll} S \to F\, I\, e & F\, \varphi\, x \to F\, (H\, \varphi)\, (c\, x) & I\, x \to x \\ F\, \varphi\, x \to \varphi\, x & H\, \varphi\, y \to a\, (\varphi\, (b\, y)) & \end{array}$$

where $F : (o \to o) \to o \to o$, $H : (o \to o) \to o \to o$ and $I : o \to o$.

Take a rewrite rule $F\, \overline{x} \to s$ for the non-terminal F. If the head symbol of s is a terminal symbol a, then we say that it is an (F, a)-rule; otherwise it is an (F, ϵ)-rule. We say that a word-language recursion scheme is *deterministic* just if for every non-terminal F and $\xi \in \Sigma \cup \{ \epsilon \}$, there is at most one (F, ξ)-rule; further, for each F, if there is a (F, ϵ)-rule, then there can be no (F, a)-rule where $a \in \Sigma$. Thus both schemes in Example 8 are non-deterministic. However there are deterministic recursion schemes that generate the languages of the Example.

Example 9. The following deterministic recursion scheme generates the language $\{ a^n\, b^n\, c^n \mid n \geq 0 \}$.

$$\begin{array}{ll} S \to e & A\, \varphi_1\, \varphi_2 \to b\, (\varphi_1\, (\varphi_2\, e)) \\ S \to a\, (A\, I\, (F\, c\, I)) & F\, \varphi_1\, \varphi_2\, x \to \varphi_1(\varphi_2\, x) \\ A\, \varphi_1\, \varphi_2 \to a\, (A\, (F\, b\, \varphi_1)\, (F\, c\, \varphi_2)) & I\, x \to x \end{array}$$

where $A : (o \to o) \to (o \to o) \to o$, $F : (o \to o) \to (o \to o) \to o \to o$ and $I : o \to o$.

2.3 Maslov's Pushdown Hierarchy of Word Languages

In the original 1974 paper [45], Maslov mentioned in brief several formalisms that are equivalent to higher-order pushdown automata. In a subsequent paper in 1976 [46], he set out the details of one such formalism, called *higher-order indexed grammars*. Using a key operation of "raising a language to a power given by another language", higher-order indexed grammars generalise Aho's indexed grammars [4] (which are the order-2 indexed grammars) to all finite orders; further, infinite-order indexed grammars define exactly the recursively enumerable languages. In a different direction, Damm [18,19] studied a system of recursion schemes that are constrained by *derived types*. Damm and Geordt [19,21] showed that, as generators of word languages, order-n safe recursion schemes are equivalent to order-n pushdown automata, which are in turn equivalent to order-n indexed grammars. The notion of safety is introduced in Section 2.4.

Theorem 10 (Maslov 1976, Damm 1982, Damm and Goerdt 1986). *For each $n \geq 0$, the following formalisms define the same class of word languages:*

(i) order-n pushdown automata
(ii) order-n indexed grammars
(iii) order-n safe recursion schemes.

The low-order languages of Maslov's pushdown hierarchy are well-known and much studied. The order-2 languages, which are indexed languages, were a topic of interest in the 1970s and early 1980s [30,27,56,23]. A notable recent advance [60] was Parys' pumping lemma for the class of ϵ-closure of order-n pushdown graphs, for every $n \geq 0$. (The result was subsequently extended by Kartzow and Parys [35] to a pumping lemma for the hierarchy of ϵ-closure of collapsible pushdown graphs.) Another significant development was the result, due to Inaba and Maneth at FSTTCS 2008 [33], that every language in the pushdown hierarchy (equivalently the OI hierarchy [19]) is context sensitive. By considering word languages in the image of iterated composites of macro tree transducers, they show that languages of the pushdown hierarchy are in non-deterministic linear space and NP-complete [33, Corollary 9].

An interesting and challenging problem is to find higher-order analogues of Parikh's Lemma and such powerful characterisation results as the Myhill-Nerode Theorem. There is a famous logical characterisation of regular languages due to Büchi: the order-0 languages are exactly those definable by S1S, monadic second-order logic with one-successor. Lautemann et al. [44] extended the characterisation to context-free languages by augmenting S1S with a notion of quantification over matchings. To our knowledge, no such logical characterisations are known for languages of order 2 or higher.

Much of what is known about the complexity of languages recognisable by automata with higher-order stacks (and other auxiliary memory) is due to Engelfriet. His seminal paper [25] contains a wealth of results. We pick out a few here that are particularly useful for studying the algorithmics of infinite structures generated by higher-order PDA.

Theorem 11 (Engelfriet 1991). *Let $s(n) \geq \log(n)$.*

(i) For $k \geq 0$, the word acceptance problem of non-deterministic order-k PDA with a two-way work-tape with $s(n)$ space is k-EXPTIME complete.

(ii) For $k \geq 1$, the word acceptance problem of alternating order-k PDA with a two-way work-tape with $s(n)$ space is $(k-1)$-EXPTIME complete.
(iii) For $k \geq 0$, the word acceptance problem of alternating order-k PDA is k-EXPTIME complete.
(iv) For $k \geq 1$, the emptiness problem of non-deterministic order-k PDA is $(k-1)$-EXPTIME complete.

2.4 The Hierarchy of Higher-Order Pushdown Trees

Fix a ranked alphabet Σ with $m := \max\{\, \mathsf{ar}(f) \mid f \in \Sigma \,\}$. The branch language [13] of a Σ-labelled ranked tree t is a set of finite and infinite words that represent its maximal branches (or paths). Writing $[m] = \{\, 1, \cdots, m \,\}$, the *branch language* of $t : dom(t) \to \Sigma$ consists of:

- infinite words $(f_1, d_1)(f_2, d_2)\cdots$ such that there exists $d_1\, d_2 \cdots \in [m]^\omega$ with $t(d_1 \cdots d_i) = f_{i+1}$ and $d_{i+1} \leq \mathsf{ar}(f_{i+1})$ for every $i \geq 0$, and
- finite words $(f_1, d_1)\cdots(f_n, d_n)\, f_{n+1}$ such that there exists $d_1 \cdots d_n \in [m]^*$ with $t(d_1 \cdots d_i) = f_{i+1}$ for every $0 \leq i \leq n$, $d_i \leq \mathsf{ar}(f_i)$ for every $1 \leq i \leq n$, and $\mathsf{ar}(f_{n+1}) = 0$.

For example, the branch language of the tree generated by the recursion scheme G_1 of Example 6 is $\{\, (f,2)^\omega \,\} \cup \{\, (f,2)^n\, (f,1)\, (g,1)^n\, a \mid n \geq 0 \,\}$. It follows from the definition that two ranked trees are equal if and only if they have the same branch language.

Definition 12. (i) Let $\mathcal{S} = \langle \Gamma, AStore_\Gamma, Op, top, \bot \rangle$ be an abstract store system. A *tree $\mathcal{S}$-automaton* is a tuple $\mathcal{A} = \langle \mathcal{S}, Q, \Sigma, \delta, q_I \rangle$ where Q is a finite set of states, $q_I \in Q$ is the initial state, Σ is a ranked alphabet, and

$$\delta \,:\, Q \times \Gamma \longrightarrow (Q \times Op \;\cup\; \{\, (f, q_1 \cdots q_{\mathsf{ar}(f)}) \mid f \in \Sigma, q_i \in Q \,\})$$

is a transition function. A *configuration* is either a pair (q, s) where $q \in Q$ and $s \in AStore_\Gamma$, or a triple of the form $(f, q_1 \cdots q_{\mathsf{ar}(f)}, s)$ where $f \in \Sigma$ and $q_1 \cdots q_{\mathsf{ar}(f)} \in Q^*$. Let $\overline{\Sigma}$ be the label-set $\{\, (f,i) \mid f \in \Sigma, 1 \leq i \leq \mathsf{ar}(f) \,\} \cup \{\, a \in \Sigma \mid \mathsf{ar}(a) = 0 \,\}$. We define the labelled transition relation between configurations induced by δ:

$$\begin{array}{ll}
(q,s) \xrightarrow{\epsilon} (q', \theta(s)) & \text{if } \delta(q, top\, s) = (q', \theta) \\
(q,s) \xrightarrow{\epsilon} (f, \overline{q}, s) & \text{if } \delta(q, top\, s) = (f, \overline{q}) \text{ and } \mathsf{ar}(f) \geq 1 \\
(q,s) \xrightarrow{a} (a, \epsilon, s) & \text{if } \delta(q, top\, s) = (a, \epsilon) \text{ and } \mathsf{ar}(a) = 0 \\
(f, \overline{q}, s) \xrightarrow{(f,i)} (q_i, s) & \text{for every } 1 \leq i \leq \mathsf{ar}(f)
\end{array}$$

Let w be a finite or infinite word over the alphabet $\overline{\Sigma}$. We say that w is a *trace* of $\mathcal{A}$ just if there is a possibly-infinite sequence of transitions $(q_I, \bot) \xrightarrow{\ell_1} \gamma_1 \cdots \xrightarrow{\ell_m} \gamma_m \xrightarrow{\ell_{m+1}} \cdots$ such that $w = \ell_1 \ell_2 \cdots$. We say that $\mathcal{A}$ *generates* the Σ-labelled tree t just in case the branch language of t coincides with the set of traces of $\mathcal{A}$.

(ii) In case $\mathcal{S} = \langle \Gamma, n\text{-}Stack_\Gamma, Op_n, top_1, \bot_n \rangle$ is the system of n-stacks over Γ, we refer to a tree $\mathcal{S}$-automaton as an *order-n pushdown tree automaton*, and simply specify it by the tuple $\langle \Gamma, Q, \Sigma, \delta, q_I \rangle$. For $n \geq 0$, we write $PushdownTree_n^\Sigma$ for the class of Σ-labelled ranked trees generated by order-n tree pushdown automata.

Example 13. The tree of Example 6 is generated by the order-1 pushdown tree automaton $\langle \{ Z, \bot \}, \{ q_I, q_1, q_2 \}, \Sigma, \delta, q_I \rangle$ where δ is defined as follows:

$$\begin{array}{lll} (q_I, \bot) \mapsto (f, q_1\, q_2) & (q_2, \bot) \mapsto push_1^Z \,;\, (f, q_1\, q_2) & (q_2, Z) \mapsto push_1^Z \,;\, (f, q_1\, q_2) \\ (q_1, \bot) \mapsto (a, \epsilon) & (q_1, Z) \mapsto pop_1 \,;\, (g, q_1). & \end{array}$$

In a FoSSaCS 2002 paper [36], Knapik, Niwiński and Urzyczyn introduced the class $SafeRecTree_n^\Sigma$ of Σ-labelled trees generated by order-n recursion schemes that are *homogeneously typed* and satisfy a syntactic constraint called *safety*. They proved that for every $n \geq 0$, $SafeRecTree_n^\Sigma = PushdownTree_n^\Sigma$. Thus $SafeRecTree_0\Sigma$, the order-0 trees, are the regular trees (i.e. trees generated by finite-state transducers), and $SafeRecTree_1\Sigma$, the order-1 trees, are those generated by order-1 pushdown tree automata.

Later in the year, in an MFCS 2002 paper [12], Caucal introduced a hierarchy of trees and a dual hierarchy of graphs, which are defined by mutual recursion, using a pair of powerful functors.[3] The functor from graphs to trees is the unravelling operation, and the functor from trees to graphs is given by inverse rational mapping. Level 0 of the graph hierarchy are the finite. Caucal showed [12, Theorem 3.5] that $SafeRecTree_n^\Sigma = CaucalTree_n^\Sigma$, where $CaucalTree_n^\Sigma$ consists of Σ-labelled trees obtained from the regular Σ-labelled-trees (i.e. trees from $PushdownTree_0^\Sigma$) by iterating n-times the operation of inverse deterministic rational mapping followed by unravelling. To summarise:

Theorem 14. *For all $n \geq 0$, $SafeRecTree_n^\Sigma = PushdownTree_n^\Sigma = CaucalTree_n^\Sigma$.*

2.5 Homogeneous Types and the Safety Constraint

Safety [36] is a syntactic constraint on the rewrite rules of a recursion scheme. It specifies whether a formal parameter of a rewrite rule may occur in a subterm of the RHS of the rule, depending on the position of the subterm, and the respective orders of the parameter and the subterm.

Definition 15. (i) A type $A_1 \to \cdots \to A_n \to o$ is *homogeneous* just if each A_i is homogeneous, and $ord(A_1) \geq ord(A_2) \geq \cdots \geq ord(A_n)$. (It follows that the base type o is homogeneous.) A term (or a rewrite rule or a recursion scheme) is *homogeneously typed* just if all types that occur in it are homogeneous.

(ii) A rewrite rule $F\, y_1 \cdots y_k \to t$ is *safe* just if for each subterm s of t that occurs in the operand position (i.e. as the second argument) of an application, for every $1 \leq j \leq k$, if the parameter y_j occurs in s then $ord(s) \leq ord(y_j)$.

[3] The functors preserve the decidability of MSO theories. It follows that all structures in the two hierarchies have decidable MSO theories.

(iii) A recursion scheme is safe just if it is homogeneously typed and all its rewrite rules are safe.

For example, the order-2 rule $F\,\varphi\,x\,y \;\rightarrow\; f\,(F\,\underline{(F\,\varphi\,y)}\,y\,(\varphi\,x))\,a$, where $F : (o \rightarrow o) \rightarrow o \rightarrow o \rightarrow o$ and $f : o \rightarrow o \rightarrow o$, is unsafe, because the order-0 parameter y occurs in the underlined order-1 subterm, which is in an operand position. Note that it follows from the definition that order-0 and order-1 recursion schemes are always safe. Safety may be regarded as a reformulation of Damm's *derived types* [19]; see de Miranda's thesis [47] for a proof of equivalence.

Remark 16. The definition of safe recursion schemes by Knapik et al. [36] assumes that all types are homogeneous. In an unpublished note, Blum and Broadbent [7] have shown that (i) homogeneity is superfluous for safety, in the sense that for generating ranked trees, homogeneously-typed safe recursion schemes are equi-expressive as safe recursion schemes, (ii) homogeneity is superfluous in general i.e. homogeneously-typed unsafe recursion schemes are equi-expressive as unsafe recursion schemes.

What is the point of safety? Though somewhat unnatural as a syntactic constraint, safety does have a clear algorithmic value. It is a well-known fact of symbolic logic and formal systems such as the lambda calculus that *capture-permitting substitution is unsound.* Therefore, when performing substitution, one must avoid variable capture, for example, by renaming bound variables. Remarkably, in safe recursion schemes, it is a relatively straightforward consequence of the definition that capture-permitting substitution *is* sound; in other words, it is safe *not* to rename bound variables. It follows that no fresh names are needed when computing the value tree of a safe recursion scheme. Knapik et al. [36] used this fact in their proof of the decidability of the MSO theories of the value trees of safe recursion schemes.

In view of their algorithmic advantage (namely, space efficiency), it is pertinent to ask if safe recursion schemes are less expressive. (We consider this important question in Section 4.3.) The safe lambda calculus [8,6] is a formalisation of safety as a subsystem of the simply-typed lambda calculus. Blum and Ong [8] showed that, using Church numerals, the numeric functions representable by simply-typed safe lambda-terms are exactly the multivariate polynomials. This should be contrasted with the classical result of Schwichtenberg [65]: the numeric functions representable by simply-typed lambda-terms are the multivariate polynomials augmented with conditionals. For a systematic study of the safe lambda calculus, including its expressivity, complexity and semantics, see Blum's doctoral thesis [6].

3 Model Checking Higher-Order Recursion Schemes

What classes of infinite structures have decidable monadic second-order (MSO) theories? One of the best known examples of such a decidable class are the *regular trees* as studied by Rabin in a landmark paper in 1969 [63]. Muller and

Shupp [48] subsequently proved that the *configuration graphs of pushdown systems* also have decidable MSO theories. In the 1990s, as finite-state technologies matured, researchers embraced the challenges of software and hence infinite-state model checking. A highlight in this period was Caucal's result [11] that *prefix-recognisable graphs* (equivalently the ϵ-closure of configuration graphs of pushdown systems) have decidable MSO theories. Another concerned *algebraic trees*, which are trees generated by context-free tree grammars. Courcelle [15] showed that these trees have decidable MSO theories. In 2002, work by Knapik et al. [36] and Caucal [12] significantly extended and unified earlier developments.

Theorem 17 (Knapik et al. 2002 & Caucal 2002). *For each $n \geq 0$, all trees in $SafeRecTree_n^{\Sigma} = PushdownTree_n \Sigma = CaucalTree_n^{\Sigma}$ have decidable MSO theories.*

We give an outline of the proof [36] which is by induction on n. Given an order-$(n+1)$ safe recursion scheme G, consider an associated tree, call it $\daleth_G$, which is obtained by contracting all the order-1 β-redexes in the rewrite rules of G. The tree $\daleth_G$ coincides with the tree generated by an order-n recursion scheme G^α i.e. $\daleth_G = [\![G^\alpha]\!]$; further the MSO theory of the original order-$(n+1)$ tree $[\![G]\!]$ is reducible to that of the order-n tree $[\![G^\alpha]\!]$ i.e. there is a computable transformation of MSO sentences $\varphi \mapsto \varphi'$ such that $[\![G]\!] \vDash \varphi$ iff $[\![G^\alpha]\!] \vDash \varphi'$ [36, Theorem 3.3]. Thanks to the safety assumption, it is sound to contract β-redexes using *capture-permitting* substitution i.e. without renaming bound variables. It follows that one can construct the tree $\daleth_G$ using only the original variables of the recursion scheme G. The same construction on an arbitrary recursion scheme would require an infinite set of fresh variable names.

3.1 Some Key Questions

Assuming safety, recursion schemes have decidable MSO theories, and their expressivity is characterised by higher-order pushdown automata. Yet safety is an awkward condition, both syntactically and semantically. Is safety really essential for these desirable properties? We consider a number of key questions.

Question 1 *MSO Decidability*: Do trees generated by recursion schemes have decidable MSO theories?

Question 2 *Automata-Theoretic Characterisation*: Find a class of automata that characterise the expressive power of recursion schemes. Specifically, can higher-order pushdown automata be so extended that they define the same class of ranked trees as recursion schemes?

Question 3 *Graph Families*: Is there a good definition of graphs generated by recursion schemes? We expect the unravelling of these graphs to coincide with the value trees of recursion schemes. What logical theories of these graphs are decidable?

Question 4 *Expressivity*: Does safety really constrain expressivity? Are there *inherently unsafe* word languages / trees / graphs?

The rest of the section is devoted to Question 1. We consider the remaining questions in Section 4.

3.2 MSO Decidability and a Semantic Proof

A first partial answer to Question 1 was obtained by Aehlig et al. [2]; they showed that all trees up to order 2, whether safe or not, and whether homogeneously typed or not, have decidable MSO theories. Independently, Knapik et al. obtained a somewhat sharper result [37]. Subsequently, in a LICS 2006 paper [53], Ong answered the question fully.

Theorem 18 (Ong 2006). *For each $n \geq 0$ the modal mu-calculus model checking problem for trees generated by order-n recursion schemes is n-*EXPTIME *complete.*

Since MSOL and the modal mu-calculus are equi-expressive over trees, it follows that these trees have decidable MSO theories. In the following we sketch a proof of the theorem.

Thanks to Emerson and Jutla [24], we can reduce the mu-calculus model checking problem to an alternating parity tree automaton (APT) acceptance problem: *Given an order-n recursion scheme G and an APT $\mathcal{B}$, does $\mathcal{B}$ have an accepting run-tree over the value tree $[\![G]\!]$?* The proof has two main ingredients.

I. The first is a *transference principle* from the value tree to an auxiliary *computation tree*, which is regular. Using game semantics [32], we establish a strong correspondence between *paths* in the value tree and *traversals* in the computation tree. This allows us to prove that the APT $\mathcal{B}$ has an accepting run-tree over the value tree if and only if it has an accepting *traversal-tree* over the computation tree.

II. The second ingredient is the simulation of an accepting traversal-tree by a certain set of annotated paths over the computation tree: we construct a *traversal-simulating* APT, $\widehat{\mathcal{B}}$, as a recognising device for this set of paths.

Thus we have

$$
\begin{aligned}
& \text{APT } \mathcal{B} \text{ has an accepting run-tree over the value tree } [\![G]\!] \\
\iff & \{ \textit{ I. Transference Principle: Traversal-Path Correspondence} \} \\
& \text{APT } \mathcal{B} \text{ has an accepting } \textit{traversal-tree} \text{ over the } \textit{computation tree } \lambda(G) \\
\iff & \{ \textit{ II. Simulation of Traversals by Paths } \} \\
& \text{Traversal-simulating APT } \widehat{\mathcal{B}} \text{ has an accepting run-tree over } \lambda(G)
\end{aligned}
$$

which is decidable, because the computation tree $\lambda(G)$ is regular.

We elaborate on the key ideas behind the proof. By construction, the value tree is the *extensional* outcome of a potentially infinite process of rewriting. It would be quite futile to analyse the value tree directly, since it has no useful structure for our purpose. Our approach is to consider what we call the *long transform*, of a recursion scheme G, written $\overline{G}$, which is obtained by expanding the right-hand side of each rewrite rule to its η-long form, inserting explicit application operators, and then currying the rule. The transform allows us to tease out the two constituent actions of the rewriting process, namely, *unfolding* and β-reduction, and hence analyse them separately. See Example 19(i). Thus, take a recursion scheme G.

- We first build an auxiliary *computation tree* $\lambda(G)$ which is obtained by performing all of the unfolding but none of the β-reduction in the rewrite rules of G. Formally $\lambda(G)$ is defined to be the value tree of the long transform $\overline{G}$, which is an order-0 recursion scheme by construction. See Example 19(ii). As no substitution is performed, no variable-renaming is needed.
- We then analyse the β-reductions *locally* (i.e. without the *global* operation of substitution) using game semantics [32].

Example 19. Fix a ranked alphabet $\Sigma = \{\, g : 2, h : 1, a : 0 \,\}$.

(i) We present the long transform $\overline{G_3}$ of an order-2 (unsafe) recursion scheme G_3:

$$G_3 : \begin{cases} S \to H\, a \\ H\, z \to F\, (g\, z) \\ F\, \varphi \to \varphi\, (\varphi\, (F\, h)) \end{cases} \mapsto \ \overline{G_3} : \begin{cases} S \to \lambda.@\, H\, (\lambda.a) \\ H \to \lambda z.@\, F\, (\lambda y.g\, (\lambda.z)\, (\lambda.y)) \\ F \to \lambda\varphi.\varphi\, (\lambda.\varphi\, (\lambda.@\, F\, (\lambda x.h\, (\lambda.x)))) \end{cases}$$

The value tree $[\![G_3]\!]$ is the tree on left in Figure 1. The only infinite path in the tree is 221^ω.

(ii) The computation tree $\lambda(G_3)$ of G_3 is the tree on the right in Figure 1. In the figure, for ease of reference, we give nodes of $\lambda(G_3)$ numeric names (in square-brackets).

(iii) With reference to Figure 1, the maximal traversals (pointers omitted) over the computation tree $\lambda(G_3)$ are:

0 1 2 3 4 5 13 14 15 16 19 20
0 1 2 3 4 5 13 14 17 18 6 7 13 14 15 16 19 20
0 1 2 3 4 5 13 14 17 18 6 7 13 14 17 18 8 9 10 21 11 12 $\cdots$

They correspond respectively to the paths $g\ a$, $g\ g\ a$ and $g\ g\ h^\omega$ in $[\![G]\!]$.

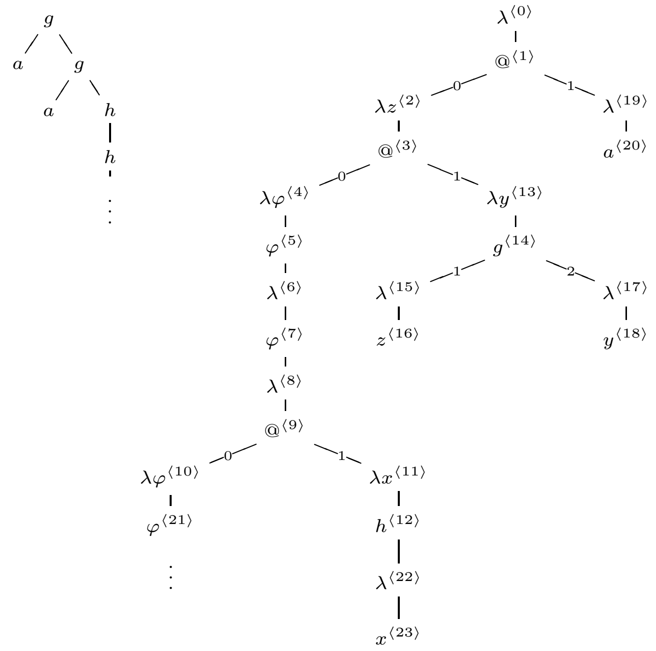
Note that we do not (need to) assume that the recursion scheme G is safe. We can now state a strong correspondence between *paths* in the value tree and *traversals* [53,54] over the computation tree. See Example 19(iii).

Theorem 20 (Path-Traversal Correspondence). *Fix a ranked alphabet Σ. Let G be an order-n recursion scheme.*

(i) There is a 1-1 correspondence between maximal paths p in the value tree $[\![G]\!]$ and maximal traversals t_p over the computation tree $\lambda(G)$.
(ii) Further, for each p, we have $p \upharpoonright \Sigma = t_p \upharpoonright \Sigma$.

The theorem is proved using game semantics [54,32]. In the language of game semantics, paths in the value tree correspond exactly to P-views in the strategy-denotation of the recursion scheme; a traversal is then (a representation of) the *uncovering*[4] of such a play. The path-traversal correspondence allows us to

[4] In game semantics [32], plays in the composite strategy $\sigma\,;\tau : A \to C$ are constructed from those in $\sigma : A \to B$ and $\tau : B \to C$ by "parallel composition plus hiding" [31]. By construction, every play s in the composite strategy $\sigma\ ;\tau$ is obtained from an *interaction sequence* $\widehat{s}$ — which is a certain sequence of moves-with-pointer from arenas A, B and C — by hiding all moves of B. We call $\widehat{s}$ the *uncovering* of s.



g
a
g
a
h
h
$\lambda^{\langle 0\rangle}$
$@^{\langle 1\rangle}$
0
1
$\lambda z^{\langle 2\rangle}$
$\lambda^{\langle 19\rangle}$
$@^{\langle 3\rangle}$
$a^{\langle 20\rangle}$
0
1
$\lambda\varphi^{\langle 4\rangle}$
$\lambda y^{\langle 13\rangle}$
$\varphi^{\langle 5\rangle}$
$g^{\langle 14\rangle}$
1
2
$\lambda^{\langle 6\rangle}$
$\lambda^{\langle 15\rangle}$
$\lambda^{\langle 17\rangle}$
$\varphi^{\langle 7\rangle}$
$z^{\langle 16\rangle}$
$y^{\langle 18\rangle}$
$\lambda^{\langle 8\rangle}$
$@^{\langle 9\rangle}$
0
1
$\lambda\varphi^{\langle 10\rangle}$
$\lambda x^{\langle 11\rangle}$
$\varphi^{\langle 21\rangle}$
$h^{\langle 12\rangle}$
$\lambda^{\langle 22\rangle}$
$x^{\langle 23\rangle}$

Fig. 1. The value tree $[\![G_3]\!]$ and computation tree $\lambda(G_3)$ of Example 19

prove that a given alternating parity tree automaton (APT) has an accepting run-tree over the value tree if and only if it has an accepting *traversal-tree* over the computation tree.

Our problem is then reduced to finding an effective method of recognising certain sets of infinite traversals over a given computation tree that satisfy the parity condition. This requires a new idea as a traversal is most unlike a path; it can jump all over the tree and may even visit certain nodes infinitely often. Our solution again exploits the game-semantic connexion. It is a property of traversals that their *P-views* are paths (in the computation tree). This allows us to simulate a traversal over a computation tree by the P-views of its prefixes, which are annotated paths of a certain kind in the same tree. The simulation is made precise in the notion of *traversal-simulating* APT. We establish the correctness of the simulation by proving that a given APT $\mathcal{B}$ has an accepting traversal-tree over the computation tree if and only if the associated *traversal-simulating* APT $\widehat{\mathcal{B}}$ has an accepting run-tree over the computation tree. The control-states of a traversal-simulating APT are *variables profiles*, which are assertions concerning variables about the APT-state being simulated and the largest priority encountered during a relevant part of the computation.

Decidability of the modal mu-calculus model checking problem for trees generated by recursion schemes follows at once since computation trees are regular, and the APT acceptance problem for regular trees is decidable [63,24].

Finally, to prove n-EXPTIME decidability of the model checking problem, we first establish a certain *succinctness property* for traversal-simulating APT: If a traversal-simulating APT has an accepting run-tree, then it has one with a reduced branching factor. The desired time bound is then obtained by analysing the complexity of solving an associated (finite) acceptance parity game, which is an appropriate product of the traversal-simulating APT and a finite deterministic graph that unravels to the computation tree in question.

3.3 Other Proofs of the Decidability Theorem

Several other proofs of Theorem 18 have been published.

In a LICS 2008 paper [29], Hague et al. gave a proof by reducing the modal mu-calculus model checking of recursion schemes to the solution of parity games over the configuration graphs of collapsible pushdown automata (see Theorem 32 in the sequel). The proof of correctness of the scheme-to-automaton transformation (Theorem 24) uses game semantics.

In a LICS 2009 paper [40], Kobayashi and Ong gave a proof that uses type theory. They exhibit a transformation that, given an APT $\mathcal{A}$, constructs a type system $\mathcal{K}_{\mathcal{A}}$ such that a recursion scheme is typable in $\mathcal{K}_{\mathcal{A}}$ if and only if the value tree of the recursion scheme is accepted by the APT $\mathcal{A}$. The model checking problem is thus reduced to a type inference problem. An advantage of this model checking algorithm is that it has an improved parameterised complexity: the time complexity is polynomial in the size of the recursion scheme, assuming that the types and the APT are fixed.

Salvati and Walukiewicz [64] recently gave yet another proof of Theorem 18. They consider trees generated by the λY-calculus, and use Krivine machine as a model of computation. Their proof is by reducing the problem of solving a parity game over the configurations of a Krivine machine to that of solving a finite parity game.

4 Collapsible Pushdown Automata

In this section, we consider Questions 2, 3 and 4 of Section 3.1. We introduce a variant of higher-order pushdown automata, called *collapsible pushdown automata*, and show that they generate the same class of trees as recursion schemes. We then define the configuration graphs of collapsible pushdown systems, and state a result on the solution of parity games over these graphs. Finally we discuss recent progress on the Safety Conjecture.

4.1 Collapsible Pushdown Automata

Collapsible pushdown automata (CPDA) are a variant of higher-order pushdown automata in which every symbol in the stack has a link to a prefix of the stack. In addition to the higher-order stack operations $push_i$ and pop_i, CPDA have an important operation called *collapse*, whose effect is to "collapse" the stack s

to the prefix indicated by the link from the $top_1(s)$. The main result is that for every $n \geq 0$, order-n recursion schemes and order-n CPDA are equi-expressive as generators of ranked trees.

Let Γ be a stack alphabet and $n \geq 1$. An *order-n collapsible stack s* is an order-n stack such that every non-$\bot$ symbol that occurs in it has a link to a collapsible stack (of order k) situated below it in s; we call the link a *$(k+1)$-link*. Note that k is necessarily less than n. We shall abbreviate order-n collapsible stack to n-stack, whenever it is clear form the context. The *empty k-stack* is defined as before. When displaying examples of n-stacks, we shall omit $\bot$ and 1-links (i.e. links to stack symbols) to avoid clutter; thus we write $[[\,][a\,b]]$ instead of $[[\bot\; a\; b]]$

For $n \geq 2$ the set $Op_n^\dagger$ of *order-n operations on collapsible stacks* consists of the following four types of operations:

(i) pop_k for each $1 \leq k \leq n$
(ii) $collapse$
(iii) $push_1^{a,k}$ for each $1 \leq k \leq n$ and each $a \in (\Gamma \setminus \{\bot\})$
(iv) $push_j$ for each $2 \leq j \leq n$.

The operation pop_i is defined as before in Section 2.1. Let s be an n-stack and $2 \leq i \leq n$. To construct $push_1^{a,i}\, s$, we first attach a link from a fresh copy of a to the $(i-1)$-stack that is immediately below the top $(i-1)$-stack of s, and then push the symbol-with-link onto the top 1-stack of s. As for *collapse*, suppose the top_1-symbol of s has a link to a (particular occurrence of) k-stack u in s. Then *collapse* s causes s to "collapse" to the prefix s_0 of s such that $top_{k+1}\, s_0 = u$. Finally, for $j \geq 2$, the *order-j push* operation, $push_j$, simply takes a stack s and duplicates the top $(j-1)$-stack of s, preserving its link structure.

Example 21. Take the 3-stack $s = [[[a]]\,[[\,][a]]]$. We have

$$push_1^{b,2}\, s = [[[a]]\,[[\,][a\,b]]] \qquad collapse\,(push_1^{b,2}\, s) = [[[a]]\,[[\,]]]$$

$$\underbrace{push_1^{c,3}(push_1^{b,2}\, s)}_{\theta} = [[[a]]\,[[\,][a\,b\,c]]]$$

Then $push_2\,\theta$ and $push_3\theta$ are respectively

$$[[[a]]\,[[\,][a\,b\,c][a\,b\,c]]] \quad \text{and} \quad [[[a]]\,[[\,][a\,b\,c]]\,[[\,][a\,b\,c]]].$$

We have $collapse\,(push_2\,\theta) = collapse\,(push_3\,\theta) = collapse\,\theta = [[[a]]]$.

As in the case of order-n stacks, the tuple $\langle \Gamma, n\text{-}Stack_\Gamma^\dagger, Op_n^\dagger, top_1, \bot_n \rangle$ is called the *system of order-n collapsible stacks*, which is an abstract store system. We shall use automata equipped with order-n collapsible stacks to define word languages and trees, and (in Section 4.2) infinite graphs.

Definition 22. Let $\mathcal{S} = \langle \Gamma, n\text{-}Stack^{\dagger}_{\Gamma}, Op^{\dagger}_{n}, top_1, \bot_n \rangle$ be the system of order-n collapsible stacks over Γ. An *order-n collapsible pushdown word-language automaton* is just a word-language $\mathcal{S}$-automaton $\langle \mathcal{S}, Q, \Sigma, \Delta, q_I, F \rangle$, and we specify it as $\langle \Gamma, Q, \Sigma, \Delta, q_I, F \rangle$. Similarly an *order-n collapsible pushdown tree automaton* is just a tree $\mathcal{S}$-automaton $\langle \mathcal{S}, Q, \Sigma, \delta, q_I \rangle$, and we specify it as $\langle \Gamma, Q, \Sigma, \delta, q_I \rangle$.

Example 23 (Urzyczyn 2003; Aehlig, de Miranda and Ong 2005).

(i) We define the language U over the alphabet $\{ \mathbf{(}, \mathbf{)}, * \}$ as follows. A *U-word* is composed of 3 segments:

$$\underbrace{\mathbf{(} \cdots \mathbf{(} \cdots \mathbf{(}}_{A} \; \underbrace{\mathbf{(} \cdots \mathbf{)} \cdots \mathbf{(} \cdots \mathbf{)}}_{B} \; \underbrace{* \cdots *}_{C}$$

- Segment A is a prefix of a well-bracketed word that ends in $\mathbf{(}$, and the opening $\mathbf{(}$ is not matched in the (whole) word.
- Segment B is a well-bracketed word.
- Segment C has length equal to the number of $\mathbf{(}$ in A.

It is a consequence of the definition that every U-word has a unique decomposition. For example, $\mathbf{(\,()\,(}\underline{\mathbf{(\,)\,(\,(\,))}}* * *$ is in U; its B-segment is underlined. For each $n \geq 0$, the word $\mathbf{(\,(}^{n}\,\mathbf{)}^{n}\,\mathbf{(}\ *^{n}\ **$ is in U, the respective B-segments are all empty.

(ii) The language U is recognisable by a *deterministic* order-2 collapsible pushdown automaton $\langle \{ q_I, q_1, q_2 \}, \{ \mathbf{(}, \mathbf{)}, * \}, \{ \bot, Z \}, \delta, q_I, \{ q_2 \} \rangle$, where $\delta : Q \times \Sigma \times \Gamma \rightarrow Op_n^{*} \times Q$ is as follows:

$$\begin{array}{ll}
(q_I, \mathbf{(}, \bot) \mapsto (push_2 \,; push_1^{Z}, q_1) & (q_1, *, Z) \mapsto (collapse, q_2) \\
(q_1, \mathbf{(}, Z) \mapsto (push_2 \,; push_1^{Z}, q_1) & (q_2, *, Z) \mapsto (pop_2, q_2) \\
(q_1, \mathbf{)}, Z) \mapsto (pop_1, q_1) &
\end{array}$$

The idea is that the pair $(q_1, Z) \in Q \times \Gamma$ indicates that the number of "$\mathbf{(}$" read, minus the number of "$\mathbf{)}$" read, is at least one. Note that $(q_1, \bot)$ indicates a "stuck configuration" which is reachable upon reading e.g. $\mathbf{(\,)}$. To illustrate, we present the computation of the U-word $\mathbf{(\,(\,)\,(\,(\,)}\ * * *$ in Figure 2. (In the figure, we omit the unimportant links.)

It follows from [3] that U is recognisable by a *non-deterministic* order-2 pushdown automaton. This illustrates the power of collapse.

The main result of this section is the following equi-expressivity result.

Theorem 24 (Hague, Murawski, Ong and Serre 2008). *For every $n \geq 0$, order-n recursion schemes and order-n collapsible pushdown tree automata define the same class of Σ-labelled trees.*

The proof is in the long version of [29]. Here we explain the main ideas.

From CPDA to Recursion Schemes. We construct an algorithm that transforms a given order-n CPDA to an equivalent order-n recursion scheme. To illustrate,

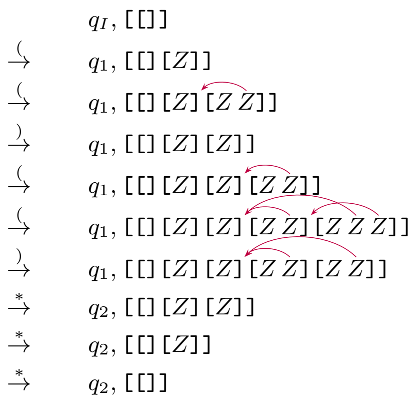

q_I, [[]]
$\xrightarrow{(}$ q_1, [[][Z]]
$\xrightarrow{(}$ q_1, [[][Z][$Z\,Z$]]
$\xrightarrow{)}$ q_1, [[][Z][Z]]
$\xrightarrow{(}$ q_1, [[][Z][Z][$Z\,Z$]]
$\xrightarrow{(}$ q_1, [[][Z][Z][$Z\,Z$][$Z\,Z\,Z$]]
$\xrightarrow{)}$ q_1, [[][Z][Z][$Z\,Z$][$Z\,Z$]]
$\xrightarrow{*}$ q_2, [[][Z][Z]]
$\xrightarrow{*}$ q_2, [[][Z]]
$\xrightarrow{*}$ q_2, [[]]

Fig. 2. The computation of the U-word **(()(()** $***$

we present the algorithm in the order-2 case, which is due to Knapik et al. [37]. For a generalisation to all finite orders, see [29]. Fix an order-2 CPDA $\mathcal{A}$ with state-set $\{0, \cdots, m-1\}$. Let 0 be the base type; we define $n+1 := n^m \to n$ where $A^m \to B$ is a shorthand for the type $A \to \cdots \to A \to B$ (m occurrences of A). For each stack symbol Z and each state $0 \le p \le m-1$, we introduce a non-terminal

$$\mathcal{F}_p^Z \; : \; 0^m \to 1^m \to 0^m \to 0$$

that represents the (top-of-stack) symbol Z with a (order-2) link in state p. In addition, for $i \in \{0, 1\}$, we introduce a non-terminal $\Omega_i : i$, and fix a start symbol $S : 0$. Let $P(i)$ be a term with an occurrence of (the parameter) i; we write $\langle P(i) \mid i \rangle$ as a shorthand for the sequence $P(0) \cdots P(m-1)$. For example $\langle \mathcal{F}_i^Z \mid i \rangle$ denotes the sequence $\mathcal{F}_0^Z \cdots \mathcal{F}_{m-1}^Z$.

We briefly explain the idea of the translation. The intuition is that a term of the form $\mathcal{F}_p^Z \, \overline{L} \, \overline{M_1} \, \overline{M_0} \; : \; 0$ represents a configuration (p, s) where p is a state and s is a 2-stack; equivalently the sequence $\langle \mathcal{F}_i^Z \, \overline{L} \, \overline{M_1} \, \overline{M_0} \mid i \rangle$ represents the 2-stack s (the state is "abstracted" by the sequence). Further

- $(p, \mathit{top}_1\, s)$ — where the top_1-symbol of s is Z which has a link to the 1-stack represented by $\overline{L} : 0^m$ — is represented by $\mathcal{F}_p^Z \, \overline{L} : 2$
- $(p, \mathit{top}_2\, s)$ is represented by $\mathcal{F}_p^Z \, \overline{L} \, \overline{M_1} : 1$
- $(p, \mathit{pop}_2\, s)$ is represented by $M_{0,p} : 0$ and $(p, \mathit{pop}_1\, s)$ by $M_{1,p} \, \overline{M_0} : 0$
- $(p, \mathit{collapse}\; s)$ is represented by $L_p : 0$.

Definition 25. The order-2 *recursion scheme determined by* $\mathcal{A}$, written $G_{\mathcal{A}}$, has the following rewrite rules. We use vector notation; for example $\overline{\psi_1}$ is a shorthand for the sequence $\psi_{1,0} \cdots \psi_{1,m-1}$.

- Start rule: $S \to \mathcal{F}_0^{\perp} \, \overline{\Omega_0} \, \overline{\Omega_1} \, \overline{\Omega_0}$

- For each $(p, Z, q, \theta) \in \delta$: $\mathcal{F}_p^Z \, \overline{\varphi} \, \overline{\psi_1} \, \overline{\psi_0} \ \rightarrow \ \Xi_{(q,\theta)}$ where $\Xi_{(q,\theta)}$ is defined by

(q,θ)	$\Xi_{(q,\theta)}$
$(q, push_1^Y)$	$\mathcal{F}_q^Y \, \overline{\psi_0} \, \langle \mathcal{F}_i^Z \, \overline{\varphi} \, \overline{\psi_1} \mid i \rangle \, \overline{\psi_0}$
$(q, push_1)$	$\mathcal{F}_q^Z \, \overline{\varphi} \, \langle \mathcal{F}_i^Z \, \overline{\varphi} \, \overline{\psi_1} \mid i \rangle \, \overline{\psi_0}$
$(q, push_2)$	$\mathcal{F}_q^Z \, \overline{\varphi} \, \overline{\psi_1} \, \langle \mathcal{F}_i^Z \, \overline{\varphi} \, \overline{\psi_1} \, \overline{\psi_0} \mid i \rangle$

(q,θ)	$\Xi_{(q,\theta)}$
(q, pop_1)	$\psi_{1,q} \, \overline{\psi_0}$
(q, pop_2)	$\psi_{0,q}$
$(q, collapse)$	φ_q

- For each $(p, Z, f, \overline{q}) \in \delta$: $\mathcal{F}_p^Z \, \overline{\varphi} \, \overline{\psi_1} \, \overline{\psi_0} \ \rightarrow \ f \, (\mathcal{F}_{q_1}^Z \, \overline{\varphi} \, \overline{\psi_1} \, \overline{\psi_0}) \cdots (\mathcal{F}_{q_{\mathsf{ar}(f)}}^Z \, \overline{\varphi} \, \overline{\psi_1} \, \overline{\psi_0})$.

Example 26. Consider the language U and the order-2 CPDA defined in Example 23. Applying the transformation, we obtain an order-2, deterministic, unsafe recursion scheme over the ranked alphabet $\{ \mathbf{(} : 1, \mathbf{)} : 1, * : 1, e : 0 \}$ that generates U.

$$\begin{array}{rcl}
S & \rightarrow & \mathcal{F}_0^{\perp} \ \overline{\Omega_0} \ \overline{\Omega_1} \ \overline{\Omega_0} \\
\mathcal{F}_0^{\perp} \ \overline{z} \ \overline{\varphi} \ \overline{x} & \rightarrow & \mathbf{(} \ \ \big(\mathcal{F}_1^Z \ \langle F_i^{\perp} \ \overline{z} \ \overline{\varphi} \ \overline{x} \mid i \rangle \ \langle F_i^{\perp} \ \overline{z} \ \overline{\varphi} \mid i \rangle \ \langle F_i^{\perp} \ \overline{z} \ \overline{\varphi} \ \overline{x} \mid i \rangle\big) \\
\mathcal{F}_1^{Z} \ \overline{z} \ \overline{\varphi} \ \overline{x} & \rightarrow & \mathbf{(} \ \ \big(\mathcal{F}_1^Z \ \langle F_i^{Z} \ \overline{z} \ \overline{\varphi} \ \overline{x} \mid i \rangle \ \langle F_i^{Z} \ \overline{z} \ \overline{\varphi} \mid i \rangle \ \langle F_i^{Z} \ \overline{z} \ \overline{\varphi} \ \overline{x} \mid i \rangle\big) \\
\mathcal{F}_1^{Z} \ \overline{z} \ \overline{\varphi} \ \overline{x} & \rightarrow & \mathbf{)} \ \ (\varphi_1 \ \overline{x}) \\
\mathcal{F}_1^{Z} \ \overline{z} \ \overline{\varphi} \ \overline{x} & \rightarrow & * \ \ z_2 \\
\mathcal{F}_2^{Z} \ \overline{z} \ \overline{\varphi} \ \overline{x} & \rightarrow & * \ \ x_2 \\
\mathcal{F}_2^{\perp} \ \overline{z} \ \overline{\varphi} \ \overline{x} & \rightarrow & e
\end{array}$$

From Recursion Schemes to CPDA. Our proof uses the theory of traversals [54,6], which is based on game semantics [32].

Let G be an order-n recursion scheme. From the computation tree $\overline{G}$, we define a labelled directed graph $\mathrm{Gr}(G)$, which will serve as a blueprint for the definition of $\mathsf{CPDA}(G)$, the CPDA determined by G. To construct $\mathrm{Gr}(G)$, we first take the forest consisting of the abstract syntactic tree of the right-hand sides of $\overline{G}$. We orient the edges towards the leaves. For each node with label f (say), the outgoing edges are labelled with directions $1, 2, \cdots, \mathsf{ar}(f)$ respectively, except that edges from nodes labelled by @ are labelled from 0. Let us write $v = E_i(u)$ just if (u, v) is an edge labelled by i. Next, for every non-terminal F, we identify the root rt_F of the abstract syntactic tree of the right-hand side of the rule for F with all nodes labelled F (which were leaves in the forest). We designate the node rt_S, where S is the start symbol of $\overline{G}$, as the root of $\mathrm{Gr}(G)$. The graph $\mathrm{Gr}(G_3)$ for the order-2 recursion scheme G_3 of Example 19 is shown in Figure 3.

We are now ready to describe $\mathsf{CPDA}(G)$. The set of nodes of $\mathrm{Gr}(G)$ will become the stack alphabet of $\mathsf{CPDA}(G)$. The initial configuration will be the n-stack $push_1^{v_0,1} \perp_n$, where v_0 is the root of $\mathrm{Gr}(G)$. For ease of explanation, we define the transition map δ as a function that takes a node $u \in \mathrm{Gr}(G)$ to a sequence of stack operations, by a case analysis of the label l_u of u. When l_u is not a variable, the action is just $push_1^{v,1}$, where v is an appropriate successor of the node u. Precisely, we define v as follows.

$$v \ := \begin{cases} E_0(u) \text{ if } l_u = @ \\ E_1(u) \text{ if } l_u = \lambda\overline{\varphi} \\ E_i(u) \text{ if } l_u \in \Sigma \end{cases}$$

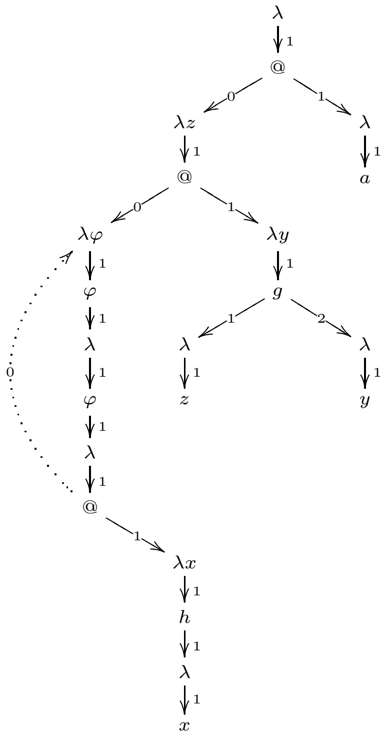


Fig. 3. The graph $\mathrm{Gr}(G_3)$ determined by the order-2 recursion scheme G_3

where i is the direction that the automaton is to explore in the value tree.

Finally, suppose l_u is a variable φ_i and its binder is a lambda node $\lambda\overline{\varphi}$ which is in turn a j-child.

- If φ has order $l \geq 1$ we define

$$\delta(u) \;:=\; \begin{cases} push_{n-l+1} \;;\; pop_1^{p+1} \;;\; push_1^{E_i(top_1),n-l+1} & \text{if } j = 0 \\ push_{n-l+1} \;;\; pop_1^{p} \;;\; collapse \;;\; push_1^{E_i(top_1),n-l+1} & \text{otherwise} \end{cases}$$

where $push_1^{E_i(top_1),k}$ is defined to be the operation $s \mapsto push_1^{E_i(top_1\, s),k}\, s$.

- If φ has order 0 we define

$$\delta(u) \;:=\; \begin{cases} pop_1^{p+1} \;;\; push_1^{E_i(top_1),1} & \text{if } j = 0 \\ pop_1^{p} \;;\; collapse \;;\; push_1^{E_i(top_1),1} & \text{otherwise.} \end{cases}$$

It can be shown that the runs of $\mathsf{CPDA}(G)$ are in 1-1 correspondence with traversals, in the sense of Ong [53] (see the systematic treatment [54]). Since traversals are *uncoverings* [32] of paths in the value tree $[\![G]\!]$, for every order-n recursion scheme G, the order-n CPDA-transform, $\mathsf{CPDA}(G)$, generates the value tree $[\![G]\!]$.

Remark 27. (i) Since the construction of the CPDA-transform in the scheme-to-automata translation is based on game semantics [32], Theorem 24 also gives an automata-theoretic characterisation of innocent strategies. There are several machine-theoretic representations of innocent game semantics in the literature: PAM, the Pointer Abstract Machine of Danos et al. [22], may be viewed as an implementation of linear head reduction of lambda-terms; Curien and Herbelin [16,17] used abstract Böhm trees as the basis of a family of abstract machines. Compared to these machines, the characterisation by CPDA seems clearly syntax-independent: contrast, for example, the type-theoretic notion of order with the order of collapsible stacks. (ii) Blum and Broadbent [7] recently proved that if the recursion scheme G is safe, then the CPDA-transform, $\mathsf{CPDA}(G)$, does not use *collapse* in its computation.

In a LICS 2012 paper [10], Carayol and Serre gave a syntactic proof of Theorem 24. Their scheme-to-CPDA translation does not use game semantics. They show that if the recursion scheme is safe, then the CPDA-translate does not use *collapse* in its computation.

4.2 Parity Games over CPDA Configuration Graphs

Definition 28. (i) Let $\mathcal{S} = \langle \Gamma, \mathit{AStore}_\Gamma, \mathit{Op}, \mathit{top}, \bot \rangle$ be an abstract store system. An *$\mathcal{S}$-transition system* is a tuple $\mathcal{T} = \langle \mathcal{S}, Q, \Delta, q_I \rangle$ where Q is a finite set of control-states, $q_I \in Q$ is the initial state, and $\Delta \subseteq Q \times \Gamma \times Q \times \mathit{Op}$ is the transition relation. A *configuration* is a pair (q, s) where $q \in Q$ and $s \in \mathit{AStore}_\Gamma$; and $(q_I, \bot)$ is the initial configuration. The transition relation Δ induces a (labelled) transition relation between configurations according to the rule: $(q, s) \xrightarrow{(q', \theta)} (q', \theta(s))$ provided $(q, \mathit{top}(s), q', \theta) \in \Delta$. The *configuration graph* of $\mathcal{T}$ is a directed graph whose vertices are the configurations, and edge-set is the induced transition relation.

(ii) In case $\mathcal{S} = \langle \Gamma, n\text{-}\mathit{Stack}_\Gamma, \mathit{Op}_n, \mathit{top}_1, \bot_n \rangle$ is the system of order-n stacks over Γ, we refer to a $\mathcal{S}$-transition system $\mathcal{T} = \langle \mathcal{S}, Q, \Delta, q_I \rangle$ as an *order-n pushdown system* (order-n PDS).

(iii) Similarly, in case $\mathcal{S} = \langle \Gamma, n\text{-}\mathit{Stack}^\dagger_\Gamma, \mathit{Op}^\dagger_n, \mathit{top}_1, \bot_n \rangle$ is the system of order-n collapsible stacks over Γ, we call a $\mathcal{S}$-transition system $\mathcal{T} = \langle \mathcal{S}, Q, \Delta, q_I \rangle$ as an *order-n collapsible pushdown system* (order-n CPDS).

Example 29 (An undecidable CPDS graph). Take the order-2 CPDS with state-set $\{ 0, 1, 2 \}$, stack alphabet $\{ a, b, \bot \}$ and transition relation given by

$$\{ (0, -, 1, t),\ (1, -, 0, a),\ (1, -, 2, b),\ (2, \dagger, 2, 1),\ (2, \dagger, 0, 0) \}$$

where $-$ means any symbol, $\dagger$ means any non-$\bot$ symbol, and $t, a, b, 0$ and 1 are shorthand for the stack operations push_2, $\mathit{push}_1^{a,2}$, $\mathit{push}_1^{b,2}$, *collapse* and pop_1 respectively. We present its configuration graph (with edges labelled by stack operations only) in Figure 4.

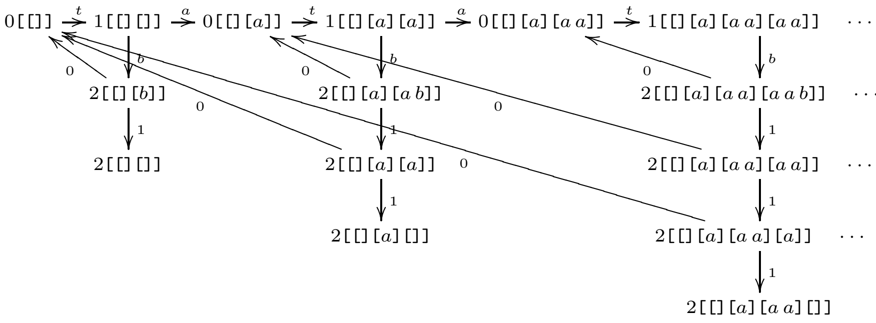


Fig. 4. A order-2 CPDS graph with an undecidable MSO theory

Let $G = \langle V, E \rangle$ be the configuration graph of an $\mathcal{S}$-transition system $\mathcal{A}$, and $Q_{\mathbf{E}} \cup Q_{\mathbf{A}}$ be a partition of Q, and let $\Omega : Q \to \{\, 0, \cdots M-1 \,\}$ be a priority function. Together they define a partition $V_{\mathbf{E}} \cup V_{\mathbf{A}}$ of V whereby a vertex belongs to $V_{\mathbf{E}}$ if and only if its control state belongs to $Q_{\mathbf{E}}$, and a priority function $\Omega : V \to \{\, 0, \cdots, M-1 \,\}$ where a vertex is assigned the priority of its control state. We call the structure $\mathcal{G} = \langle G, V_{\mathbf{E}}, V_{\mathbf{A}} \rangle$ an *order-n CPDS game graph* and the pair $\mathbb{G} = \langle \mathcal{G}, \Omega \rangle$ an *order-n CPDS parity game.*

In this section we consider the problem:

($\mathbf{P}_1$) *Given an order-n CPDS parity game decide if Éloïse has a winning strategy from the initial configuration.*

The Problem ($\mathbf{P}_1$) is closely related to the following problems:

($\mathbf{P}_2$) *Given an order-n CPDS graph G, and a modal mu-calculus formula φ, does φ hold at the initial configuration of G?*

($\mathbf{P}_3$) *Given an APT and an order-n CPDS graph G, does the APT accept the unravelling of G?*

($\mathbf{P}_4$) *Given an MSO formula φ and an order-n CPDS graph G, does φ hold at the root of the unravelling of G?*

Using the techniques of Emerson and Jutla [24], it is straightforward to show that Problem ($\mathbf{P}_1$) is polynomially equivalent to Problems ($\mathbf{P}_2$) and ($\mathbf{P}_3$); and Problem ($\mathbf{P}_1$) is equivalent to Problem ($\mathbf{P}_4$) – the reduction from ($\mathbf{P}_1$) to ($\mathbf{P}_4$) is polynomial, but non-elementary in the other direction.

A useful fact is that the unravelling of an order-n CPDS graph is actually generated by an order-n collapsible pushdown tree automaton (putting labels on the edges makes the order-n CPDS graph *deterministic* and hence its unravelling as desired). Thus an important consequence of Theorem 24 is the following.

Proposition 30. *Let t be the value tree of an order-n recursion scheme. Consider the following problems:*

($\mathbf{P}'_2$) *Given t and a modal mu-calculus formula φ, does φ hold at the root of t?*

($\mathbf{P}'_3$) *Given t and an APT, does the automaton accept t?*

($\mathbf{P}'_4$) *Given t and an MSO formula φ, does φ hold at the root of t?*

Then problem $(\mathbf{P}_i)$ *is polynomially equivalent to problem* $(\mathbf{P}'_i)$ *for* $i = 2, 3$ *and* 4.

Since the modal mu-calculus model checking problem for value trees of recursion schemes is decidable [53], we obtain the following as an immediate consequence.

Theorem 31. $(\mathbf{P}_1)$, $(\mathbf{P}_2)$, $(\mathbf{P}_3)$ *and* $(\mathbf{P}_4)$ *are n-EXPTIME complete.*

Another consequence of Theorem 24 is that it gives new techniques for model checking or solving games played on infinite structures generated by automata. In particular it leads to new proofs/optimal algorithms for the special cases that have been considered previously [67,66,37]. Conversely, as Theorem 24 works in both directions, we note that a solution of Problem $(\mathbf{P}_1)$ would give a new proof of the decidability of Problems $(\mathbf{P}'_2)$, $(\mathbf{P}'_3)$ and $(\mathbf{P}'_4)$, and would give a new approach to problems on recursion schemes. Actually, the techniques of Walukiewicz [67] and Knapik et al. [37] can be generalised to solve order-n CPDS parity games without reference to Ong's work [53]. Further they give effective winning strategies for the winning player (which was not the case in [37] where the special case $n = 2$ was considered).

Theorem 32 (Hague, Murawski, Ong and Serre 2008). *The problem of solving an order-n CPDS parity game is n-EXPTIME complete. Further one can build an order-n collapsible pushdown transducer (i.e. automaton with output) that realises a winning strategy for the winning player.*

Remark 33. This result can be generalised to the case where the game has an arbitrary ω-regular winning condition, and is played on the ϵ-closure of the configuration graph of an order-n CPDS graph. Consequently parity games on Caucal graphs [12,66] are a special case of this problem.

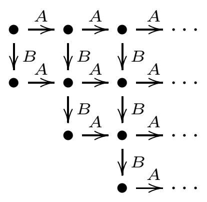
The Caucal graphs have decidable MSO theories [12]. Do the configuration graphs of CPDS also have decidable MSO theories?

Theorem 34 (Hague, Murawski, Ong and Serre 2008). *There is an order-2 CPDS whose configuration graph has an undecidable MSO theory. Hence the class of* ϵ*-closure of configuration graphs of CPDS strictly contains the Caucal graphs.*

For a proof, recall that MSO interpretation preserves MSO decidability. Now consider the following MSO interpretation I of the configuration graph of the order-2 CPDS in Example 29:

$$I_A(x, y) = x \xrightarrow{C} y \wedge x \xrightarrow{R} y \qquad I_B(x, y) = x \xrightarrow{1} y$$

with $C = \overline{1}^* \, \overline{b} \, a \, t \, b \, 1^*$ and $R = 0 \, t \, a \, \overline{0} \vee \overline{1} \, 0 \, t \, a \, \overline{0} \, 1$.



Note that for the A-edges, the constraint C requires that the target vertex should be in the next column to the right, while R specifies the correct row. Observe that I's image is the "infinite half-grid" which has an undecidable MSO theory.

Winning Regions and Logical Reflection Broadbent et al. [9] gave the first characterisation of the winning regions of order-n CPDS parity games: they are regular sets defined by a new class of automata. As a corollary, it is shown that recursion schemes are *reflective* with respect to MSOL and modal mu-calculus i.e. there is an algorithm that transforms a given property φ and a recursion scheme G to a new recursion scheme G^{φ} that *reflects* the property in that $[\![G^{\varphi}]\!]$ has the same underlying tree as $[\![G]\!]$ except that the nodes that satisfy φ have a special label. Thus we may view G^{φ} as a transform of G that can internally observe its behaviour against a specification φ.

4.3 Expressivity and the Safety Conjecture

The Safety Conjecture [36] is about the expressivity of *safe* recursion schemes. The conjecture states that there is an *inherently unsafe tree* i.e. there is a tree which is generated by an unsafe recursion scheme, but not by any safe recursion scheme. In view of Theorems 17 and 24, the Safety Conjecture can be stated equivalently in terms of CPDA. As recursion schemes (and CPDA) can be used to define word languages, trees and graphs, it is meaningful to consider the expressivity of safe recursion schemes in each of the three cases.

Aehlig et al. [3] showed that there are no inherently unsafe order-2 word languages: for every unsafe order-2 recursion scheme (respectively order-2 CPDA), there is a safe non-deterministic order-2 recursion scheme (respectively order-2 PDA) that defines the same language. However, Parys [58] has recently shown that the conjecture holds for word languages when restricted to deterministic devices.

Theorem 35 (Parys 2011). *There is a language (similar to U of Example 23) which is recognised by a deterministic order-2 CPDA but not by any deterministic order-n PDA, for any $n \geq 0$.*

The Safety Conjecture (for trees) was recently proved by Parys [59].

Theorem 36 (Parys 2012). *There is a tree which is generated by an order-2 unsafe recursion scheme but not by any safe order-n recursion scheme, for any $n \geq 0$.*

It follows from Theorem 34 that the Safety Conjecture is false in the case of graphs.

5 Application to Model Checking Higher-Order Functional Programs

In a POPL 2009 paper [39], Kobayashi proposed a type-based model checking algorithm for recursion schemes. He considered properties expressible by *trivial automata* [1], which are Büchi tree automata in which every state is final (thus the acceptance condition is trivial). The key result is that given a trivial automaton

A, there is an intersection type system $\mathcal{T}_A$ such that for every recursion scheme G, the automaton A accepts the value tree $[\![G]\!]$ if and only if G is typable in $\mathcal{T}_A$. Thus model checking is reduced to type inference. (This type-based approach was subsequently extended by Kobayashi and Ong [40] to a model checking algorithm for all alternating parity tree automata.) Even though trivial automata correspond to a tiny fragment of the modal mu-calculus, the model checking problem is hugely expensive: the complexity remains n-EXPTIME hard [42]. Kobayashi showed that many verification problems of higher-order functional programs, such as reachability, flow analysis and resource usage verification, can be expressed as trivial automata model checking problems of recursion schemes. Thus, the model checking algorithm can serve as a basis for the verification of higher-order functional programs. In a subsequent paper [38], Kobayashi presented a "practical" type inference algorithm. He showed that a tool implementation of the algorithm, called TRecS, performs remarkably well on a range of small but tricky examples, despite the hyper-exponential worst-case complexity.

There has been much progress in higher-order model checking in recent years. Kobayashi [41] and Neatherway, Ramsay and Ong [50] have introduced algorithms for model checking recursion schemes against trivial automata which are inspired by or based on game semantics. There have also been advances in the automatic safety verification of realistic classes of functional programs. Ong and Ramsay [55] have proposed an extension of recursion schemes, called *pattern matching recursion schemes*, which model algebraic data types and function definition by pattern matching, features that are ubiquitous in functional programs. Using counterexample-guided abstraction refinement (CEGAR), they have proposed a sound and semi-complete method for verifying pattern-matching recursion schemes. In a different direction, Kobayashi et al. [43] have formalised predicate abstraction and CEGAR for higher-order model checking, which enable the automatic verification of programs that use infinite data domains such as integers.

6 Conclusions

Higher-order model checking is challenging and worthwhile. Recursion schemes and collapsible pushdown automata are robust and highly expressive higher-order formalisms for constructing infinite structures. They have rich algorithmic properties. Recent progress in the theory have used semantic methods (such as game semantics and types) as well as automata-theoretic techniques from algorithmic verification. Despite prohibitive worst-case complexity, there are "practical" model checking algorithms which perform remarkably well on small but tricky examples.

Acknowledgements. Much of my work presented here is based on collaboration. I am grateful to all my collaborators for their help. Thanks are also due to EPSRC for financial support.

References

1. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. Logical Methods in Computer Science 3, 1–23 (2007)
2. Aehlig, K., de Miranda, J.G., Ong, C.-H.L.: The Monadic Second Order Theory of Trees Given by Arbitrary Level-Two Recursion Schemes Is Decidable. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 39–54. Springer, Heidelberg (2005)
3. Aehlig, K., de Miranda, J.G., Ong, C.-H.L.: Safety Is not a Restriction at Level 2 for String Languages. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 490–504. Springer, Heidelberg (2005)
4. Aho, A.: Indexed grammars - an extension of context-free grammars. J. ACM 15, 647–671 (1968)
5. de Bakker, J.W., de Roever, W.P.: A calculus for recursive program schemes. In: ICALP, pp. 167–196 (1972)
6. Blum, W.: The Safe Lambda Calculus. Ph.D. thesis, University of Oxford (2008)
7. Blum, W., Broadbent, C.: The CPDA-transform of a safe recursion scheme does not collapse (2009) (in preparation)
8. Blum, W., Ong, C.H.L.: The safe lambda calculus. LMCS 5(1) (2009)
9. Broadbent, C.H., Carayol, A., Ong, C.H.L., Serre, O.: Recursion schemes and logical reflection. In: LICS, pp. 120–129 (2010)
10. Carayol, A., Serre, O.: Collapsible pushdown automata and labeled recursion schemes: Equivalence, safety and effective selection. In: LICS, pp. 165–174 (2012)
11. Caucal, D.: On Infinite Transition Graphs Having a Decidable Monadic Theory. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 194–205. Springer, Heidelberg (1996)
12. Caucal, D.: On Infinite Terms Having a Decidable Monadic Theory. In: Diks, K., Rytter, W. (eds.) MFCS 2002. LNCS, vol. 2420, pp. 165–176. Springer, Heidelberg (2002)
13. Courcelle, B.: Fundamental properties of infinite trees. TCS 25, 95–169 (1983)
14. Courcelle, B.: Recursive applicative program schemes. In: Handbook of Theoretical Computer Science, vol. B, pp. 459–492. MIT Press (1990)
15. Courcelle, B.: The monadic second-order logic of graphs IX: machines and their behaviours. Theoretical Computer Science 151, 125–162 (1995)
16. Curien, P.L., Herbelin, H.: Computing with abstract Böhm trees. In: Fuji International Symposium on Functional and Logic Programming, pp. 20–39. World Scientific (1998)
17. Curien, P.L., Herbelin, H.: Abstract machines for dialogue games, coRR abs/0706.2544 (2007)
18. Damm, W.: Higher type program schemes and their tree languages. Theoretical Computer Science, pp. 51–72 (1977)
19. Damm, W.: The IO- and OI-hierarchy. TCS 20, 95–207 (1982)
20. Damm, W., Fehr, E., Indermark, K.: Higher type recursion and self-application as control structures. In: Neuhold, E. (ed.) Formal Descriptions of Programming Concepts, pp. 461–487. North-Holland, Amsterdam (1978)
21. Damm, W., Goerdt, A.: An automata-theoretical characterization of the OI-hierarchy. Information and Control 71, 1–32 (1986)
22. Danos, V., Herbelin, H., Regnier, L.: Game semantics and abstract machines. In: LICS, pp. 394–405 (1996)
23. Duske, J., Parchmann, R.: Linear indexed languages. TCS 32, 47–60 (1984)

24. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: FOCS, pp. 368–377 (1991)
25. Engelfriet, J.: Interated stack automata and complexity classes. Information and Computation 95, 21–75 (1991)
26. Esparza, J., Schwoon, S.: A BDD-Based Model Checker for Recursive Programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
27. Gilman, R.H.: A shrinking lemma for indexed languages. TCS 163, 277–281 (1996)
28. Guessarian, I.: Algebraic Semantics. Springer (1981)
29. Hague, M., Murawski, A.S., Ong, C.H.L., Serre, O.: Collapsible pushdown automata and recursion schemes. In: LICS, pp. 452–461 (2008)
30. Hayashi, T.: On derivation trees of indexed grammars: An extension of the uvwxy-theorem. Publ. RIMS Kyoto Univ. 9, 61–92 (1983)
31. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
32. Hyland, J.M.E., Ong, C.H.L.: On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model. Information and Computation 163, 285–408 (2000)
33. Inaba, K., Maneth, S.: The complexity of tree transducer output languages. In: FSTTCS, pp. 244–255 (2008)
34. Jones, N.D., Muchnick, S.S.: Complexity of finite memory programs with recursion. Journal of the Association for Computing Machinery 25, 312–321 (1978)
35. Kartzow, A., Parys, P.: Strictness of the Collapsible Pushdown Hierarchy. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 566–577. Springer, Heidelberg (2012)
36. Knapik, T., Niwiński, D., Urzyczyn, P.: Higher-Order Pushdown Trees Are Easy. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 205–222. Springer, Heidelberg (2002)
37. Knapik, T., Niwiński, D., Urzyczyn, P., Walukiewicz, I.: Unsafe Grammars and Panic Automata. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1450–1461. Springer, Heidelberg (2005)
38. Kobayashi, N.: Model-checking higher-order programs. In: PPDP, pp. 25–36 (2009)
39. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: POPL, pp. 416–428 (2009)
40. Kobayashi, N., Ong, C.H.L.: A type theory equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: LICS, pp. 179–188 (2009)
41. Kobayashi, N.: A Practical Linear Time Algorithm for Trivial Automata Model Checking of Higher-Order Recursion Schemes. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 260–274. Springer, Heidelberg (2011)
42. Kobayashi, N., Ong, C.H.L.: Complexity of model checking recursion schemes for fragments of the modal mu-calculus. LMCS 7(4) (2011)
43. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and cegar for higher-order model checking. In: PLDI, pp. 222–233 (2011)
44. Lautemann, C., Schwentick, T., Thérien, D.: Logics for Context-Free Languages. In: Pacholski, L., Tiuryn, J. (eds.) CSL 1994. LNCS, vol. 933, pp. 205–216. Springer, Heidelberg (1995)
45. Maslov, A.N.: The hierarchy of indexed languages of an arbitrary level. Soviet Math. Dokl. 15, 1170–1174 (1974)
46. Maslov, A.N.: Multilevel stack automata. Problems of Information Transmission 12, 38–43 (1976)

47. de Miranda, J.: Structures generated by higher-order grammars and the safety constraint. Ph.D. thesis, University of Oxford (2006)
48. Muller, D.E., Schupp, P.E.: The theory of ends, pushdown automata, and second-order logic. Theoretical Computer Science 37, 51–75 (1985)
49. Murawski, A.S., Ong, C.-H.L., Walukiewicz, I.: Idealized Algol with Ground Recursion, and DPDA Equivalence. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 917–929. Springer, Heidelberg (2005)
50. Neatherway, R.P., Ramsay, S.J., Ong, C.H.L.: A traversal-based algorithm for higher-order model checking. In: ICFP, pp. 353–364 (2012)
51. Nivat, M.: Langages algébriques sur le magma libre et sémantique des schémas de programme. In: Proc. ICALP, pp. 293–308 (1972)
52. Nivat, M.: On the interpretation of recursive program schemes. Symposia Mathematica 15, 255–281 (1975)
53. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: LICS, pp. 81–90 (2006), `www.cs.ox.ac.uk/people/luke.ong/personal/publications/lics06-long.pdf`
54. Ong, C.H.L.: Local computation of beta-reduction by game semantics (2012), `www.cs.ox.ac.uk/people/luke.ong/personal/publications/locred.pdf` (preprint)
55. Ong, C.H.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: POPL, pp. 587–598 (2011)
56. Parchmann, R., Duske, J., Specht, J.: On deterministic indexed languages. Information and Control 45, 48–67 (1980)
57. Park, D.M.R.: Fixpoint induction and proofs of program properties. In: Michie, D., Meltzer, B. (eds.) Machine Intelligence, vol. 5 (1970)
58. Parys, P.: Collapse operation increases expressive power of deterministic higher order pushdown automata. In: STACS, pp. 603–614 (2011)
59. Parys, P.: On the significance of the collapse operation. In: LICS, pp. 521–530 (2012)
60. Parys, P.: A pumping lemma for pushdown graphs of any level. In: STACS, pp. 54–65 (2012)
61. Patterson, M.: Equivalence problems in a model of computation. Ph.D. thesis, University of Cambridge (1967)
62. Plotkin, G.D.: LCF as a programming language. Theoretical Computer Science 5, 223–255 (1977)
63. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Trans. Amer. Maths. Soc. 141, 1–35 (1969)
64. Salvati, S., Walukiewicz, I.: Krivine Machines and Higher-Order Schemes. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 162–173. Springer, Heidelberg (2011)
65. Schwichtenberg, H.: Definierbare funktionen im lambda-kalkul mit typen. Archiv Logik Grundlagen-forsch 17 (1976)
66. Cachat, T.: Games on Pushdown Graphs and Extensions. Ph.D. thesis, RWTH Aachen (2003), `http://www.liafa.jussieu.fr/~txc/Download/Cachat-PhD.pdf`
67. Walukiewicz, I.: Pushdown processes: games and model-checking. Information and Computation 157, 234–263 (2001)

Discrete Linear Dynamical Systems

Joël Ouaknine

Department of Computer Science, Oxford University
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
joel@cs.ox.ac.uk

Abstract. The theory of *dynamical systems* is concerned with describing and studying the evolution of systems over time, where a 'system' is represented as a vector of variables, and there is a fixed rule governing how the system evolves. Dynamical systems originate in the development of Newtonian mechanics, and have widespread applications in many areas of science and engineering. Systems that evolve in a piecewise continuous manner (typically via differential equations) are known as *continuous* dynamical systems, whereas systems exhibiting discrete transitions (commonly via difference equations) are known as *discrete* dynamical systems.

The theory of dynamical systems comprises a rich body of techniques and results, mainly geared towards the analysis of long-term qualitative behaviour of systems: existence and uniqueness of attractors, fixed points, or periodic points, sensitivity to initial conditions, etc. From a computer-science perspective, it is somewhat surprising to note that the literature is largely devoid of work on *decision problems* concerning dynamical systems, e.g., whether a fixed point or a particular region will actually be reached in finite time, whether a variable will assume negative values infinitely often, etc. Such questions, in turn, have numerous applications in a wide array of scientific areas, such as theoretical biology (analysis of L-systems, population dynamics), microeconomics (stability of supply-and-demand equilibria in cyclical markets), software verification (termination of linear programs), probabilistic model checking (reachability in Markov chains, stochastic logics), quantum computing (threshold problems for quantum automata), as well as combinatorics, term rewriting, formal languages, cellular automata, the study of generating functions, etc.

In this tutorial, I will first briefly introduce the main elements of the theory of (both continuous and discrete) dynamical systems, using several illustrative examples. I will then present various interesting decision problems, mainly in the context of discrete *linear* dynamical systems, for which there already are many open questions. Finally, I will survey some of the main known results and techniques, which draw on a wide array of mathematical tools, including linear algebra, algebraic and analytic number theory, real algebraic geometry, and model theory.

XML Schema Management: A Challenge for Automata Theory

Thomas Schwentick[⋆]

Technische Universität Dortmund, Germany

Abstract. XML is the standard format for data exchange over the internet and the Web contains millions of XML documents. Even though the structure of XML documents can be very flexible, it is desirable for many applications that documents come with a schema that describe their structure in a concise way. Whereas there exists a lot of software for the manipulation of XML documents and modern database management systems can deal with XML, the foundations of XML schema and document management systems are still not fully understood and they constitute an active research area.

Schema languages for XML, most prominently XML Schema and DTD, are closely related to concepts from Formal Language Theory, including context-free grammars, tree automata and regular expressions. However, the official standards of the *World Wide Web Consortium* (W3C) pose various restrictions that have not been much studied in Formal Language Theory before the advent of XML (and its precursor SGML). Altogether, the foundations of XML schema and document management systems raise a lot of new challenges to Automata Theory, in particular, to provide suitable algorithms and concepts.

The aim of this talk is to describe some of these challenges, to report on recent developments and to highlight some current directions of research. Topics will include the expressive power of schema languages, the repair of schemas that do not obey the restrictions posed by the W3C standards, the combination of schemas, schema conversion, schema minimisation, inference of schemas from a given set of documents, segmentation of schemas for distributed documents, and constraints that combine structural and data aspects.

[⋆] We acknowledge the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599.

On the Complexity of Shortest Path Problems on Discounted Cost Graphs[⋆]

Rajeev Alur, Sampath Kannan, Kevin Tian, and Yifei Yuan

University of Pennsylvania, Philadelphia, PA, US

Abstract. Discounted Cost Register Automata (DCRA) associate costs with strings in a regular manner using the operation of discounted sum. The min-cost optimization problem for DCRAs corresponds to computing shortest paths in graphs with more general forms of discounting than the well-studied notion of future discounting. We present solutions to two classes of such shortest path problems: in presence of both past and future discounting, we show the decision problem is NP-complete, but has a polynomial-time approximation scheme; in presence of two future discounting criteria that are composed in a prioritized manner, we show that the problem is solvable in NEXPTIME.

1 Introduction

The classical shortest path problem is to determine the minimum-cost path from a given source to a given target vertex in a finite directed graph whose edges are labeled with costs from a numerical domain, where the cost of a path is the sum of the costs of the edges it contains. In a *generalized* version of the shortest-path problem, each edge is labeled with a cost as well as a discount factor: at every step, the cost of each subsequent edge is scaled by the current discount factor (that is, the cost of a path consisting of the edges $e_1 e_2 \cdots e_n$ is given by the expression $\sum_i (c_i \prod_{j<i} d_j)$, where c_i and d_i denote respectively the cost and discount of the edge e_i) [8]. This form of *future discounting* is used in the study of Markov decision processes and more recently, in quantitative analysis of systems [6,3]. The problem of computing the shortest path in presence of such future discounting can be solved in polynomial-time [9,8].

While the existing work on generalized shortest paths considers only future discounting, there are some natural variations for associating costs with paths in presence of discounting. For example, we can define *past discounting*, where the cost of each preceding edge is scaled by the current discount factor (that is, the cost of a path is $\sum_i (c_i \prod_{j>i} d_j)$), and *global discounting*, where each cost is scaled by the discount factors of all the edges appearing in the path (that is, the cost of a path is $\sum_i (c_i \prod_j d_j)$). The goal of this paper is to initiate a systematic study of shortest path problems for such models with different notions of discounting.

⋆ This research was partially supported by NSF awards CCF 1137084, CCF 1138996, and CCF 0915777.

Our framework for defining a general class of discounted shortest-path problems is based upon the recently proposed notion of *regular functions* that map strings over an input alphabet Σ to a numerical domain with a specified set of operations [2]. For our purpose, the numerical domain $\mathbb{D}$ consists of pairs (c, d), where c ranges over incremental costs and d ranges over discount factors, $(0, 1)$ is the identity, and the binary *discounted sum* operation is defined as $(c_1, d_1) \otimes (c_2, d_2) = (c_1 + d_1 * c_2, d_1 * d_2)$. A regular function is computed by a *discounted cost register automaton* (DCRA), a deterministic machine that maps strings over an input alphabet to cost values using a finite-state control and a finite set of registers containing values in $\mathbb{D}$. At each step, the machine reads an input symbol, updates its control state, and updates its registers using expressions involving the discounted sum operation (such as $x := (c, d) \otimes x$; $x := x \otimes (c, d)$; $x := x \otimes y$; $y := (0, 1)$, where x and y are registers). After processing the input, the machine outputs the cost stored in one of the registers. The appeal for the class of functions computed by DCRAs is based on its connection to the well understood theory of regular string-to-string transformations with multiple equivalent characterizations and closure properties [5,4,1]. For example, if a function f is computable, then so is f^R defined by $f^R(w) = f(w^R)$, where w^R is the reverse of the string w; and if we were to allow a DCRA to make *speculative* decisions based on a regular property of the suffix of the input, instead of just the current input symbol, then such *regular-look-ahead* does not add to expressiveness. Different versions of discounted shortest-path problems turn out to be special cases of the *min-cost* problem for DCRAs, namely, given a function defined by a DCRA, find a string with minimal cost.

To solve the min-cost problem for DCRAs with one register, we focus on the shortest-path problem in a graph with *past-and-future discounting*: the cost of a path $e_1 e_2 \cdots e_n$ is given by $\sum_i (c_i \prod_{j<i} f_j \prod_{j>i} p_j)$, where c_i, f_i, and p_i denote, respectively, the cost, future discount, and past discount, of the edge e_i. This generalizes problems such as future discounting, past discounting, and global discounting. We show that the decision version of the problem is NP-complete: while the strongly-connected-components in the input graph can be analyzed efficiently, the problem is NP-hard even for acyclic graphs. Then, we develop a polynomial-time approximation scheme to solve the problem.

Our next set of results focus on the min-cost problem for DCRAs with two registers. We first consider the *prioritized shortest-path problem*: each edge is labeled two cost-discount pairs (c, d) and (c', d'), and the cost of a path $e_1 e_2 \cdots e_n$ corresponds to evaluating the expression $(c_1, d_1) \otimes (c_2, d_2) \cdots (c_n, d_n) \otimes (c'_1, d'_1) \otimes (c'_2, d'_2) \cdots (c'_n, d'_n)$. Thus, this corresponds to having two future-discounting functions, where the cost of a path is obtained by taking the discounted sum of a high-priority future-discounted cost with a low-priority future-discounted cost. While in the classical future discounting problems, for a given cycle, either it is beneficial to skip the cycle entirely, or it is beneficial to repeat it indefinitely, with prioritized discounting, the optimal number of times a cycle should be repeated depends on the cost/discount of the context. While the structure of an optimal path can be complex, we can establish an upper bound on the length of such a path, leading to decidability. We show that the decision version of the prioritized shortest-path problem is solvable in NEXPTIME. A prioritized shortest path problem corresponds to a DCRA with two registers x and y, where the registers are composed only at the end. If we allow repeated composition using the update $\{x := x \otimes y; y := (0, 1)\}$ on any edge, we are still able to establish decidability. Decidability of the min-cost problem for the general class of DCRAs remains an open problem.

2 Discounted Cost Register Automata

We use $\mathbb{D}$ to denote the domain consisting of pairs (c, d), where c is a cost and d is a discount factor. The typical choice for $\mathbb{D}$ is $\mathbb{D} = \mathbb{Q}^{\geq 0} \times \mathbb{Q}^{\geq 0}$, where $\mathbb{Q}^{\geq 0}$ denotes the set of nonnegative rational numbers. We define the *discounted sum* operator $\otimes$ for elements in $\mathbb{D}$ as follows. For any two elements $(c_1, d_1), (c_2, d_2)$ from $\mathbb{D}$, $(c_1, d_1) \otimes (c_2, d_2) = (c_1 + d_1 * c_2, d_1 * d_2)$. It is easy to check that $(\mathbb{D}, \otimes)$ forms a semigroup with the identity $(0, 1)$. For an element $e = (c, d) \in \mathbb{D}$, we denote the first component of e by $e.cost$ and the second component by $e.discount$, i.e. $e.cost = c$ and $e.discount = d$.

A discounted cost register automaton (DCRA) is a deterministic machine that maps strings over an input alphabet to costs using a finite-state control and a finite number of registers that hold values in $\mathbb{D}$. At every step, the machine reads an input symbol, updates its control state, and updates its registers using a parallel assignment. The right-hand-side in each assignment is an expression built from registers and constants in $\mathbb{D}$ using the discounted sum operation. The assignment is required to be *copyless*: no register appears more than once in the right-hand-sides of these assignments. The copyless restriction is common in the theory of transducers, and ensures that the output grows linearly in the size of the input. The output function associates with each accepting state an expression over the registers, evaluating which gives the resulting cost. The syntax and semantics of DCRAs is formalized in the following definitions.

A **discounted cost register automaton (DCRA)** M is a tuple $(\Sigma, Q, q_0, F, X, \delta, \rho, \mu)$, Σ is a finite input alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of accepting states, X is a finite set of registers, $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function, $\rho : Q \times \Sigma \times X \rightarrow (X \cup \mathbb{D})^*$ is the register-update function with the copyless restriction: for each state $q \in Q$, each register $x \in X$, and every symbol $a \in \Sigma$, x appears at most once in the multiset of strings $\{\rho(q, a, y) | y \in X\}$, $\mu : F \rightarrow (X \cup \mathbb{D})^*$ is the output function with the copyless restriction: for each state $q \in F$, each variable $x \in X$, x appears at most once in $\mu(q)$.

A *configuration* of a DCRA $M = (\Sigma, Q, q_0, F, X, \delta, \rho, \mu)$ is a pair (q, s), where $q \in Q$ is a state and $s : X \rightarrow \mathbb{D}$ is a valuation function that maps each register to an element in $\mathbb{D}$. The valuation function s naturally extends to a mapping from $(X \cup \mathbb{D})^*$ to $\mathbb{D}$ by evaluating the discounted sum. The run of M on an input string $w = a_1 \dots a_n \in \Sigma^*$ is a sequence of configurations $(q_0, s_0) \dots (q_n, s_n)$, where q_0 is the initial state, s_0 is the initial valuation that maps each register to the identity

$(0,1)$, and $q_{i+1} = \delta(q_i, a_{i+1})$, and for each $x \in X$, $s_{i+1}(x) = s_i(\rho(q_i, a_{i+1}, x))$, for $0 \le i < n$. The DCRA M defines a (partial) function $[\![M]\!]$ from Σ^* to $\mathbb{Q}^{\ge 0}$: if $q_n \in F$ then $[\![M]\!](w) = s_n(\mu(q_n)).cost$, and $[\![M]\!](w)$ is undefined otherwise.

It is worth noting that a DCRA is basically the same as a *streaming string transducer* (SST) that maps strings over input alphabet to strings over output alphabet [1]: a DCRA interprets string concatenation as discounted sum and hence its output symbols and register values are elements of $\mathbb{D}$ rather than strings over the output alphabet as is the case for SSTs. SSTs have the same expressiveness as MSO-definable string-to-string transformations, two-way deterministic sequential transducers, and Macro string transducers [5,4,1]. This class of *regular* string-to-string transformations has appealing closure properties such as closure under reversal and closure under regular look-ahead, and these carry over to DCRA-definable functions. In other words, DCRAs capture a robust class of functions for associating costs with strings by composing elements in $\mathbb{D}$ using the discounted sum operation in a regular manner.

The following decision problems are natural for DCRAs:

1. Given a DCRA M, and a string w over the input alphabet Σ, the *evaluation problem* is to compute the value of $[\![M]\!](w)$.
2. Given two DCRAs M_1 and M_2 over the same input alphabet Σ, the *equivalence checking problem* is to decide whether $[\![M_1]\!](w) = [\![M_2]\!](w)$ for every string w over Σ.
3. Given a DCRA M over input alphabet Σ, the *min-cost problem* is to decide the value of $\mathit{inf}\{[\![M]\!](w) | w \in \Sigma^*\}$.

It is easy to see that the evaluation problem for DCRAs is solvable in time linear in the size of input string. The equivalence checking problem for DCRAs can be solved in time polynomial in the number of states and exponential in the number of registers using the techniques discussed in [2]. In this paper we focus on the complexity of the min-cost problem for DCRAs with at most 2 registers.

3 Past-and-Future Discounting

First observe that to solve the min-cost problem for a DCRA, we can ignore the input symbols, and focus on computing shortest paths in the directed graph corresponding to the state-transition structure of a DCRA. We first look at discounted shortest-path problems corresponding to DCRAs with one register.

3.1 Generalized Shortest Path Problem

Given a DCRA M with one register x, where each update is of the form $x := x \otimes (c, d)$, the min-cost problem for M coincides with a problem called the *generalized shortest-path problem*, or shortest-paths in presence of only future-discounting, defined below.

Given a labeled directed graph $G = (V, E, L)$, where $L : E \to \mathbb{D}$ is the labeling function, and two vertices $s, t \in V$, the **generalized shortest path problem** is to find the s-t path p that minimizes the cost $L(p).cost$, where the labeling function is extended to paths using the discounted sum operator, that is, for a path $p = e_1 \ldots e_k$, $L(p) = L(e_1) \otimes L(e_2) \cdots L(e_k)$.

We denote $C(p) = L(p).cost$ and $D(p) = L(p)\ .discount$. Polynomial-time algorithms for this optimization problem are given in [9].

Theorem 1. *Given a labeled directed graph $G = (V, E, L)$, the generalized shortest path problem can be solved in $O(mn^2 \log n)$, where $n = |V|$ and $m = |E|$.*

Here, we are more interested in the structure of the optimal path. A **lasso** is a path p consisting of a simple path p_0 followed by a simple cycle l such that l is the only cycle in p. Given a lasso $p = p_0 l$ such that $D(l) < 1$, we define the limiting cost of p to be the limit of the costs of paths in the sequence $(p_0 l, p_0 ll, \ldots)$, which equals $C(p_0) + D(p_0)\frac{C(l)}{1-D(l)}$. It turns out that the optimal path must be either a simple path from s to t, or a lasso. In other words, if it is beneficial to include a cycle to reduce the cost, then it is beneficial to repeat the cycle arbitrarily many times. This property holds even when the graph has finite number of vertices, but infinitely many edges.

Lemma 1. *Let $G = (V, E)$ be a graph, with V finite but E possibly infinite. For any vertices $s, t \in V$, and any labeling function $L : E \to \mathbb{D}$ there is an optimal path which is either a simple path from s to t, or a lasso from s.*

Proof. Suppose the optimum p is not a simple path from s to t, then p has the structure $p_0 l p_1$, where l is the first cycle in p. If $C(l) + D(l)C(p_1) < C(p_1)$, we have that the limiting cost of the lasso $p_0 l$ is no more than $C(p_0 l^n p_1)$ for any natural number n. If $C(l) + D(l)C(p_1) \geq C(p_1)$, we have that $C(p_0 p_1) \leq C(p_0 l p_1)$. In either case, we can see by induction that the solution is a simple path or a lasso. □

3.2 Shortest Path in Past and Future Discounted Graphs

The generalized shortest path problem can be seen as a *future* discount problem – the discount at an edge applies to the costs of all future edges. Reversing the direction on the edges, we see that it is equivalent to a *past* discount problem, where the discount at an edge applies to all past edges. We consider the following variant: each edge e is now given a cost $c(e)$, a past discount $p(e)$ to be applied to all preceding edges, and a future discount $f(e)$ to be applied to all succeeding edges. In this problem, we assume that discounts are in the range $[0, 1]$. The cost of a path $p = e_1 \ldots e_k$ is $C(p) = \sum_{i=1}^{k} c(e_i) \prod_{j<i} f(e_j) \prod_{j>i} p(e_j)$. Given a directed graph $G = (V, E)$, vertices $s, t \in V$, and cost function $c : E \to \mathbb{Q}^{\geq 0}$ and discount functions $p, f : E \to [0, 1]$, the **past and future discount problem** seeks to find an s-t path p minimizing $C(p)$.

This variant corresponds to the DCRA with one register, where each update is of the form $x := (0, p') \otimes x \otimes (c', f')$. Note that on each edge, $c(e) = c'p', p(e) = p'$ and $f(e) = f'p'$.

First, we present a sequence of assumptions about the graph structure such that if each assumption does not hold, the optimal cost is easy to compute.

Lemma 2. *We may assume that any strongly connected component in G satisfies at least one of the following: all of the past discounts are equal to 1, or all of the future discounts are equal to 1.*

Proof. Suppose not. Then let S be a strongly connected component, and e_1 an edge with past discount < 1 and e_2 an edge with future discount < 1. Since this is a strongly connected component, there is a cycle containing e_1 and e_2. Thus, a path from s to S, arbitrarily many iterations of a fixed cycle containing e_1 and e_2, and a path to t has cost arbitrarily close to 0. □

Lemma 3. *We may assume that every non-trivial strongly connected component in G has an edge with a past discount < 1 or an edge with a future discount < 1.*

Proof. Suppose not. Then we can construct a graph G' on which this is true. Consider a strongly connected component S in G in which each edge has both discounts equal to 1. Then any optimal path p reaching S at u and leaving S at v uses a shortest path (in the classical discount-less sense) from u to v. Thus, we may replace S as follows: for a vertex $v \in S$, we replace it with vertices v_- and v_+. For an edge (u, v) with $u \notin S$, replace the endpoint v with v_-. Similarly, for (v, u) with $u \notin S$, replace v with v_+. Now, for all pairs of vertices $u, v \in S$, add an edge (u_-, v_+) with cost equal to the cost of the (classical) shortest path from u to v in S and both discounts 1. It is clear that replacing all non-trivial strongly connected components in this way yields a graph G' with each component containing at least a discount, and that a solution in G' determines a solution G and vice versa. □

Lemma 4. *We may assume there is a topological ordering of the strongly connected components of G such that all strongly connected components with a past discount less than 1 occur before components with a future discount less than 1.*

Proof. Suppose not. Then we have a path from s to a component with a future discount less than 1, to a component with a past discount less than 1, to t. If we use a cycle repeatedly in the future discount component with discount < 1 and similarly use a cycle in the past discount component, we drive the cost to 0. □

Now, we are ready to prove our results.

Theorem 2. *The decision version of the past and future discount problem (that is, given K, is there path p with $C(p) \leq K$) is* NP*-complete .*

Proof. First, the problem is NP. From Lemma 4, we know that in a topological ordering, the strongly connected components with a past discount occur before strongly connected components with a future discount. From Lemma 1, we know that the structure of an optimum subpath in the strongly connected components must be a simple path or a lasso. Thus, a sufficient proof would be the path itself.

The problem is NP-hard, by a reduction from subset product (which is shown to be NP-hard in [7]). □

Now we proceed to establish that the problem can be approximated efficiently.

Theorem 3. *The past and future discount problem has a polynomial-time approximation scheme.*

Proof. Given a graph $G = (V, E)$, cost and discount functions c, f, p, and an approximation factor ε, we will give an approximation scheme that finds a path with cost at most $(1+\varepsilon)$ times that of the minimum cost path. We will assume that G has the structure imposed by the preceding lemmas.

By Lemma 4, we note that in a topological ordering of the strongly connected components of G, all of the components with past discounts precede all of the components with future discounts. Let us denote the strongly connected components $S_1, \ldots, S_r$ (in topological sort order), and let us say that k is such that for all $i \leq k$, S_i is either trivial (consisting of a single vertex) or has a past discount, and for all $i > k$, S_i is either trivial or has a future discount. We use dynamic programming from s through S_k, and a similar, reversed process, from t backwards through S_{k+1}, and then stitch the results together.

Let $\delta = \frac{\log(1+\varepsilon)}{2n}$ where $|V| = n$, and let c be the highest cost value on any edge. At each vertex, we will store the path with the best future discount that has cost in the interval $[(1+\delta)^i, (1+\delta)^{i+1})$. Then the cost of a simple s-t path is at most $n \cdot c$, and so we need to manage at most $O\left(\frac{n \log c}{\log(1+\varepsilon)}\right)$ such buckets. Let $T_v[i]$ denote the best (least) discount for paths in the bucket $[(1+\delta)^i, (1+\delta)^{i+1})$.

We start our dynamic program at s, where we have no paths. The program then processes entire strongly connected components at a time. For a component S_i, we first consider all of the edges from outside S_i to a vertex of S_i. Let $e = (u, v)$ with $u \in S_{i'}$ for some $i' < i$ and $v \in S_i$. Then for each l that $T_u[l]$ is not empty, we look in the bucket for the interval j containing $p(e) \cdot (1+\delta)^{l+1} + c(e) \cdot T_u[l]$, and if $T_u[l] \cdot f(e)$ is less than $T_v[j]$, then we update $T_v[j]$ with this new value.

Having resolved any paths to vertices in S_i that do not use any other vertices in S_i, we now address paths to vertices in S_i through other vertices. Note that if a strongly connected component is trivial, this case does not arise. For each pair of vertices $u, v \in S_i$, and each $T_u[l]$ that is not empty, we build a graph which is just S_i, except we add a vertex t' and an edge (v, t') with cost $(1+\delta)^{l+1}$ and past discount $T_u[l]$. All other edges will have the same costs and discounts as in G. Now, since S_i had only past discounts < 1, we can solve a past discount problem to find the best (possibly infinite) path from u to t'. Note then that this represents a path from s to v, going through u, where the subpath from s to u is the one stored in $T_u[l]$. Repeating this for each l and each pair of vertices completely resolves the table T_v for each $v \in S_i$.

We can do this for all i up to k. We perform the analogous process for S_{k+1} and subsequent components: we reverse the roles of past and future discounts, and reverse the orientations on the edges. So for $v \in S_i$ for $i \leq k$, $T_v[l]$ will have the path with the best future discount in the l-th bucket, while for $v \in S_i$ for $i > k$, $T_v[l]$ will have the path with the best past discount. Now, for all edges (u, v) with $u \in S_i$ and $v \in S_j$ with $i \leq k < j$, and all l_1, l_2 for which $T_u[l_1]$ and $T_v[l_2]$ are non-empty, we consider the path adjoining the path in the l_1-th bucket at u to (u, v) to the path in the l_2-th bucket at v (note this is an s-t path). Now the claim is that the minimum cost path from across all such u, v, l_1, l_2 has cost at most $(1 + \varepsilon)$ times the true optimum.

Lemma 5. *For each $u \in \bigcup_{i \leq k} S_i$, and each l, the path from s to u realizing the discount $T_u[l]$ has cost at most $\frac{(1+\delta)^{l+1}}{1+\varepsilon}$.*

Proof. Observe that the only time a path's cost becomes overestimated is when it is placed in bucket j (representing paths with approximate costs in the interval $[(1+\delta)^j, (1+\delta)^{j+1})$), where the cost can only be overestimated by a factor $(1+\delta)$. Since a path can be put in a bucket only twice per strongly connected component, we see that its cost is overestimated by at most $(1 + \delta)^{2n} = (1 + \frac{\log(1+\varepsilon)}{2n})^{2n} \leq e^{\log(1+\varepsilon)} = 1 + \varepsilon$. □

A similar lemma applies for $u \in \bigcup_{i>k} S_i$ and t. Thus we have found the path with the least approximate cost, where for each path, the approximate cost is at most $(1 + \varepsilon)$ times its true cost, and in particular, this path has cost at most $(1 + \varepsilon)$ times the cost of the optimum path. □

4 Prioritized Discounting

In this section, we look at the prioritized discounting problem which corresponds to the min-cost problem for DCRAs with two registers where one register always leads the other in both update and output.

4.1 Shortest Path in Prioritized Discounted Graph

We first define the Prioritized Discounted Graph and the shortest path problem in a Prioritized Discounted Graph.

A **prioritized discounted graph** is a labeled directed graph $G = (V, E, L_x, L_y)$, where $L_x, L_y : E \to \mathbb{D}$ are labeling functions. Given a prioritized discounted graph $G = (V, E, L_x, L_y)$ and $s, t \in V$, the **prioritized shortest-path problem** seeks to find the s-t path p minimizing the cost defined as $C(p) = (L_x(p) \otimes L_y(p)) .cost$.

The prioritized shortest-path corresponds to the min-cost problem of DCRAs with two registers x and y, and each update is of the form $\{x := x \otimes (c_x, d_x); y := y \otimes (c_y, d_y)\}$ and the output function is $x \otimes y$.

We show that the decision version of the shortest path problem for prioritized discounted graph is decidable, assuming that $\mathbb{D} = \mathbb{Q}^{\geq 0} \times [0, 1]$.

Theorem 4. *Given a prioritized discount graph G and a nonnegative rational K, deciding whether there is an s-t path in G such that $C(p) \leq K$ is solvable in* NEXPTIME.

Proof. For a path p, let $(C_x(p), D_x(p)) = L_x(p)$ and $(C_y(p), D_y(p)) = L_y(p)$. We call cycle l a "x-neutral cycle" if $L_x(l) = (0, 1)$. If there is a path p such that

$C(p) = C_x(p) + D_x(p)C_y(p) \leq K$, then the x-cost of p (i.e. $C_x(p)$) is at most K. Let us call a path p a "candidate path" if $C_x(p) \leq K$. If there is an edge e in p such that $D_x(e) = 0$, then p has the simple structure as in the future discount problem. Therefore, we may assume G doesn't contain such edges. We first consider the case where G doesn't contain any x-neutral cycles. Let $p_0 l$ be the "best" lasso minimizing $\lim_{i\to\infty} C_x(p_0 l^i)$. If $K \geq \lim_{i\to\infty} C_x(p_0 l^i)$, $\lim_{i\to\infty} C(p_0 l^i) = \lim_{i\to\infty} C_x(p_0 l^i) \leq K$. Otherwise, the length of any candidate path cannot exceed L for some natural number L. The following lemma shows an exponential upper bound for L.

Lemma 6. *Let T be the least limiting x-cost among all the lassos. If $K < T$, the length of any candidate path is at most L, for some $L = 2^{O(nb)}$, where n is the number of vertices in G and b is the maximum number of bits to describe the labeling functions and the bound K.*

Proof. Fix a vertex v. Let m be the maximum number of occurrences of v in a candidate path. Let's consider the candidate path p with the least x-cost among all the paths in which v appears m times. Suppose $p = p_0 l_1 l_2 ... l_{m-1} p_m$. Here, p_0 is a path from s to v and l_i's are cycles that start and end in v, p_m is a path from v to t. For brevity, let $c_i = C_x(l_i)$ and $d_i = D_x(l_i)$. By Lemma 1, we further assume that p_0, p_m are simple paths and l_i's are simple cycles.

First we claim that $c_i/(1-d_i) \leq c_{i+1}/(1-d_{i+1})$, for $i = 1...m-2$. Note that we define $c_i/(1-d_i) = \infty$, if $d_i = 1$. Suppose not, there is a cycle l_i such that $c_i/(1-d_i) > c_{i+1}/(1-d_{i+1})$. This results in a path $p' = p_0 l_1 ... l_{i+1} l_i ... l_{m-1} p_m$ with the same occurrences of v and x-cost less than $C_x(p)$.

Let $Q_i = C_x(l_i l_{i+1} ... l_{m-1} p_m)$. We claim that $Q_1 > ... > Q_m$. Suppose not, we have $Q_i \leq Q_{i+1}$ for some i. Then $T \leq \lim_{n\to\infty} C_x(p_0 l_1 ... l_{i-1} l_i^n) \leq K$.

Let T_1 and T_2 be the least and second least limiting x-cost among the simple cycles that start and end in v. Let $l_1, l_2, ..l_k$ be the cycles with the limiting x-cost T_1, so the limiting x-cost of $l_i (i > k)$ is at least T_2, and $Q_1 < T_1$.Consider any $i > k$. If $d_i = 1, Q_i - Q_{i+1} = c_i + d_i \cdot Q_{i+1} - Q_{i+1} = c_i > 0$. If $d_i < 1$, $Q_i - Q_{i+1} = c_i + d_i \cdot Q_{i+1} - Q_{i+1} = c_i - (1-d_i)Q_{i+1} = (1-d_i)(\frac{c_i}{1-d_i} - Q_{i+1}) \geq (1-d) \cdot (T_2 - T_1)$. Here d is the largest x-discount among all the simple cycles other than 1. Therefore, $Q_{k+1} \geq Q_m + \delta(m-k-1)$, where $\delta = min\{c_i, (1-d)(T_2-T_1) | i = 1...m-1, c_i > 0\}$. Also we know $Q_{k+1} < T_1$, so $m < \frac{T_1}{\delta} + k + 1 = k + 2^{O(nb)}$.

Finally, we show an exponential bound for k. Since $C_x(p_0 l_1 ... l_k) \leq K$,we know

$$C_x(p_0 l_1 ... l_k) = C_x(p_0) + D_x(p_0) T_1 (1 - \prod_{i=l}^{k} d_i) \leq K$$

Therefore, $k \leq \log(1 - \frac{K - C_x(p_0)}{T_1 D_x(p_0)}) / \log d \leq \frac{K}{(C_x(p_0) + T_1 D_x(p_0) - K)\log 1/d} = 2^{O(nb)}$. Therefore, $L \leq 2^{O(nb)}$. □

Now consider the case where G contains x-neutral cycles. Let p be a path such that $C(p) \leq K$. If p does not contain any x-neutral cycles, either the best lasso has the limiting cost at most K or by Lemma 6, the length of p is at most L. Otherwise $p = p_0 l p_1$ for some x-neutral simple cycle l. Without loss of generality, assume there is no x-neutral cycles in p_0 and $C_y(lp_1) < C_y(p_1)$. Then $\lim_{i\to\infty} C(p_0 l^i p_1') \leq C(p)$, where p_1' is the path obtained by eliminating all the x-neutral cycles in p_1. By Lemma 6, the length of $p_0 l p_1'$ is at most $L + 2n$. □

We show a lower bound for the complexity of shortest path problem in a prioritized discounted graph.

Theorem 5. *Given a prioritized discounted graph G, and a source vertex s and a target vertex t, deciding whether there is a path p from s to t such that $C(p) \leq K$ is* NP*-hard.*

Proof. We reduce from subset product problem [7]. □

4.2 Shortest Path in Prioritized Past and Future Discounted Graph

Consider the most general min-cost problem for DCRAs with one register: each update is of the form $x := (c', d') \otimes x \otimes (c, d)$. In fact, these DCRAs can be modeled by DCRAs with two registers, where each update is $\{x := (c', d') \otimes x; y := y \otimes (c, d)\}$ and the output is $x \otimes y$. The min-cost problem for this set of DCRAs can be formalized as a variant shortest path problem in prioritized discounted graphs: Given a prioritized discounted graph $G = (V, E, L_x, L_y)$, and $s, t \in V$, we wish to find an s-t path minimizing $\left(L_x(p^R) \otimes L_y(p)\right).cost$. Here, p^R denote the reverse of p (that is, if $p = (e_1 e_2 ... e_k)$,$p^R = (e_k e_{k-1} ... e_1)$). By applying similar idea as that in theorem 4 and analyzing the cost of the $L_x(p^R)$, it is easy to see that this variant shortest path problem is decidable for $\mathbb{D} = \mathbb{Q}^{\geq 0} \times [0, 1]$.

Theorem 6. *Given a prioritized discounted G and a nonnegative rational K, deciding whether there is an s-t path in G such that $\left(L_x(p^R) \otimes L_y(p)\right).cost \leq K$ is solvable in* Nexptime*.*

4.3 Shortest Path in Prioritized Discounted Resetting Graph

A **prioritized discounted resetting graph** is a labeled directed graph $G = (V, E_1, E_2, L_x, L_y)$. V is the set of vertices and there are two types of edges E_1 and E_2. The labeling functions are only defined on the edges in E_1, i.e. $L_x, L_y : E_1 \to \mathbb{D}$ are the labeling functions. E_2 is the set of resetting edges. We denote $p \in E_i^*$ if the path p only consists of edges from E_i, for $i = 1, 2$. Given a prioritized discounted resetting graph $G = (V, E_1, E_2, L_x, L_y)$ and two vertices $s, t \in V$, the **shortest path problem** for G is to find the s-t path $p = (p_1 e_1 p_2 e_2 ... p_k e_k)$ which minimizes the cost defined as $C(p) = (L_x(p_1) \otimes L_y(p_1) \cdots L_x(p_k) \otimes L_y(p_k)).cost$. Here $p_i \in E_1^*, e_i \in E_2$ and we assume that every path from s to t ends with an edge in E_2 without loss of generality.

The shortest path problem for prioritized discounted resetting graphs corresponds to the min-cost problem of a subset of DCRAs, where there are two registers x, y and each update is of the form $\{x := x \otimes (c_x, d_x); y := y \otimes (c_y, d_y)\}$(we model this update as the edges in E_1) or $\{x := x \otimes y, y := (0, 1)\}$ (we model this update as the edges in E_2), and the output function is $x \otimes y$.

Now we show that the decision version of the shortest problem in a prioritized discounted resetting graph is decidable, with the assumption that $\mathbb{D} = \mathbb{Q}^{\geq 0} \times [0, 1]$.

Theorem 7. *Given a prioritized discounted resetting graph $G = (V, E_1, E_2, L_x, L_y)$, and $s, t \in V$ and a nonnegative rational $K \in \mathbb{Q}^{\geq 0}$, deciding whether there is an s-t path p in G, such that $C(p) \leq K$ is decidable.*

Proof. We reduce the shortest path problem in G to the generalized shortest path problem in a graph $G' = (V', E', L')$ with finitely many vertices but potentially infinitely many edges. First, for each edge $e \in E_2$, we construct a vertex v_e. We also create a source vertex s' and target vertex t'. Thus, $V' = \{s', t', v_e | e \in E_2\}$. Now we show the construction of the edges in G' and the labeling functions. Consider any edge $e = (u, v) \in E_2$, and any path $p \in E_1^*$ from s to u, we construct an edge $e_p = (s', v_e)$ with label $L'(e_p) = L_x(p) \otimes L_y(p)$. Consider each (ordered) pair of edges (e_1, e_2) such that $e_i = (u_i, v_i) \in E_2$ for $i = 1, 2$. For any path $p \in E_1^*$ from v_1 to u_2 we construct an edge $e_p = (v_{e_1}, v_{e_2})$ with label $L'(e_p) = L_x(e_p) \otimes L_y(p)$. Finally, for any edge $e = (v, t) \in E_2$, we construct an edge $e' = (v_e, t')$ with label $L'(e') = (0, 1)$. It is easy to see the equivalence between the shortest path in G and G'.

Lemma 7. *For any s-t path p in G, there exists an s'-t' path p' in G' such that $C(p) = C(p')$. For any s'-t' path p' in G', there exists an s-t path p in G, such that $C(p) = C(p')$.*

Let $G_1 = (V, E_1, L_x, L_y)$ and $L_x(p) \otimes L_y(p) = (C_1(p), D_1(p))$ for any path $p \in E_1^*$. We now describe a nondeterministic algorithm to solve the shortest path in the prioritized discounted resetting graph G. By Lemma 1, there is a generalized shortest path in G', which is a simple path or a lasso. First, the algorithm guesses the structure of the generalized shortest path p' in G'. If p' is a simple path, by Lemma 7, the shortest path in G has the structure $p = (p_1 e_1 \dots p_k e_k)$, where $e_i = (u_i, v_i) \in E_2$ and $p_i \in E_1^*$ for $i = 1 \dots k$ and $k \leq |E_2|$. Second, since $C(p) \leq K$, the cost of the subpath $p_1 e_1$, which is from s to v_1 through u_1, is at most K, i.e. $C_1(p_1) = C(p_1 e_1) = (L_x(p_1) \otimes L_y(p_1)).\mathit{cost} \leq K$. Therefore, the algorithm suffices to solve the prioritized shortest-path problem in G_1 with source s and target u_1 and bound K. If there is an infinite sequence of paths p'_j in G_1 with unbounded length and limiting cost at most K, then by theorem 4, $\lim_{j \to \infty} D_1(p'_j) = 0$. Therefore, the algorithm outputs "yes", since by taking any finite path p'_2 from v_1 to t, we have $\lim_{j \to \infty} C(p'_j e_1 p'_2) = \lim_{j \to \infty} C_1(p'_j) \leq K$; otherwise, there are finitely many candidate paths. Among all the candidate paths, the algorithm guesses one path p_1. Now, the algorithm will solve for the second subpath p_2 between v_1 and u_2. Note that, since $C(p_1 e_1 p_2 e_2) \leq K$, we know $C_1(p_2) = C(p_2 e_2) \leq K_1$, where $K_1 = \frac{K - C_1(p_1)}{D_1(p_1)}$. Therefore, the algorithm solves the prioritized shortest-path problem in G_1 with source v_1 and target u_2 and bound K_1. The algorithm solves other subpaths similarly. Finally, if there is a guessed path from s to t with cost at most K, output "yes", otherwise output "no". If the guessed path p' in G' is a lasso, again by Lemma 7, the corresponding shortest path in G has the structure $p = (p_1 e_1 \cdots p_j e_j \cdots p_k e_k p_{k+1} e_j \cdots p_k e_k p_{k+1} e_j \cdots)$, where $e_i = (u_i, v_i) \in E_2$ for $i = 1 \dots k$, $k \leq |E_2|$, and the cycle $l = (e_j \cdots p_k e_k p_{k+1} e_j)$ repeats after the subpath $p_0 = (p_1 e_1 \cdots p_j)$. The algorithm behaves as the same when guessing the subpath p_i between v_{i-1} and u_i. If there is a guessed path p with limiting cost $\lim_{i\to\infty} C(p_0 l^i) \leq K$, output "yes", otherwise output "no". □

5 Conclusions

The model of Discounted Cost Register Automata defines a robust class of functions for mapping strings to costs using the discounted sum operator in a regular manner. The min-cost problem for this class offers a unifying framework for generalizing the classical notion of discounting. While decidability of the min-cost problems for the general class of DCRAs remains an open problem, we have solved two interesting special cases. The shortest path problem in presence of past-and-future discounting is NP-complete with a polynomial-time approximation scheme. In prioritized discounting, two cost criteria with future discounting are combined using discounted sum. The structure of the optimal path becomes significantly more complex in this case, and we have established NEXPTIME upper bound for this problem, and also proved decidability for variants.

References

1. Alur, R., Černý, P.: Streaming Transducers for Algorithmic Verification of Single-pass List-processing Programs. In: Proc. of Principles of Programming Languages, pp. 599–610 (2011)
2. Alur, R., D'Antoni, L., Deshmukh, J.V., Raghothaman, M., Yuan, Y.: Regular functions, cost register automata, and generalized min-cost problems (2012), `http://www.cis.upenn.edu/~alur/rca12.pdf`
3. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. ACM Trans. Comput. Log. 11(4) (2010)
4. Courcelle, B.: Graph Operations, Graph Transformations and Monadic Second-Order Logic: A survey. Electronic Notes in Theoretical Computer Science 51, 122–126 (2002)
5. Engelfriet, J., Hoogeboom, H.J.: MSO definable String Transductions and Two-way Finite-State Transducers. ACM Transactions on Computational Logic 2(2), 216–254 (2001)
6. Filar, J., Vrieze, F.: Competitive Markov Decision Processes. Springer (1997)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1990)
8. Goldberg, A.V., Plotkin, S.A., Tardos, É.: Combinatorial Algorithms for the Generalized Circulation Problem. In: Foundations of Computer Science, pp. 432–443 (1988)
9. Oldham, J.D.: Combinatorial Approximation Algorithms for Generalized Flow Problems. In: Symposium on Discrete Algorithms, pp. 704–714 (1999)

Termination of Rule-Based Calculi for Uniform Semi-Unification

Takahito Aoto[1] and Munehiro Iwami[2]

[1] RIEC, Tohoku University, Japan
aoto@nue.riec.tohoku.ac.jp

[2] Dept. of Mathematics and Computer Science, Interdisciplinary Faculty of Science and Engineering, Shimane University, Japan
munehiro@cis.shimane-u.ac.jp

Abstract. Uniform semi-unification is a generalization of unification; its efficient algorithms have been extensively studied in (Kapur et al., 1994) and (Oliart&Snyder, 2004). For (uniform) semi-unification, several variants of rule-based calculi are known. But, some of these calculi in the literature lack the termination property, i.e. not all derivations are terminating. We revisit symbolic semi-unification whose solvability coincides with that of uniform semi-unification. We give an abstract criterion of the strategy on which a general rule-based calculus for symbolic semi-unification terminates. Based on this, we give an alternative and robust correctness proof of a rule-based uniform semi-unification algorithm.

Keywords: Semi-Unification, Rule-Based Calculi, Termination.

1 Introduction

Describing algorithms using rule-based calculi is often useful to present algorithms abstractly and to study the correctness of algorithms by separating the issue from those of the search strategy and/or efficiency. For many algorithms, rule-based calculi have been widely adapted and commonly used to extend or modify the algorithms; well-known such examples are the unification algorithm and the Knuth-Bendix completion algorithm (see e.g. [1]).

Semi-unification is a generalization of unification. Its application includes non-termination proving of term rewriting systems [3, 8, 16] and polymorphic type inference problems of ML languages [7, 11]; it is also related to some problems in proof theory [17] and in computational linguistics [2]. Like unification, if a semi-unifier exists, there exists a most general semi-unifier [7, 10, 17]. Unlike unification, however, semi-unification is undecidable in general [10]. Hence, many decidable classes of semi-unification have been studied: uniform semi-unification [4, 8, 15, 17–19], acyclic semi-unification [11, 14], left-linear semi-unification [6, 9], quasi-monadic semi-unification [13] and semi-unification in two variables [12].

Decidability of uniform semi-unification has been shown in various articles almost at the same time [4, 5, 8, 12, 17]. Efficient algorithms for uniform semi-unification have been extensively studied in [8, 15]. Like with unification, working on graphs is mandatory in order to give efficient algorithms, and thus these best efficient algorithms are based on graphs. The basis of correctness of these algorithms, however, is given by a simple version of the algorithms given in [8] that can be almost adapted as a rule-based calculus (not on graphs but on terms). In [15], a variant of this calculus are adapted as a rule-based calculus but no correctness proof is presented.

In contrast to similar rule-based calculi for the unification, some of such calculi in the literature lack the termination property, i.e. not all derivations are terminating. It is mentioned in [8] that "it can be shown that Algorithm A-1 will always terminate either reporting failure or semi-unifiability;..." without a proof. But, actually, derivations in their rule-based calculus terminate only if some suitable interpretation or derivation strategy is assumed. The calculus given in [15] is more flexible than that of [8], and it is not terminating either.

It may be thought that giving a terminating variant of these calculi is easy. However, it is, at least, difficult to adapt a termination proof similar to the one applied to calculi for unification for the following reasons: (1) equations in "solved" form may later be changed to "unsolved", and (2) the multiset of the (extended) variables in the equations may increase. In fact, as pointed out in [13], it seems not easy to give a well-founded ordering that guarantees the termination of such calculi.

In this paper, we revisit symbolic semi-unification, whose solvability coincides with that of uniform semi-unification. We present a new characterization of symbolic semi-unification, which reinforces the correspondence between these presentations of uniform semi-unification. We give an abstract criterion of the strategy on which a general rule-based calculus for symbolic semi-unification terminates. In this way, we give an alternative and robust proof for the correctness of a rule-based uniform semi-unification algorithm.

2 Preliminaries

We denote sets of *arity-fixed function symbols* and *variables* by $\mathcal{F}$ and $\mathcal{V}$, respectively. The set of *terms* is denote by $\mathrm{T}(\mathcal{F}, \mathcal{V})$. The set of variables in an object α, which may be a term, etc. is denoted by $\mathcal{V}(\alpha)$. A *context* is a term in $\mathrm{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V})$ containing a single occurrence of $\Box$, which is a special constant not contained in $\mathcal{F}$. A term obtained by replacing $\Box$ in a context C with a term t is denoted by $C[t]$. A term s is a *subterm* of a term t (written as $s \trianglelefteq t$) if $t = C[s]$ for some context C. A *substitution* σ is a mapping from $\mathcal{V}$ to $\mathrm{T}(\mathcal{F}, \mathcal{V})$ with the finite domain $\mathrm{dom}(\sigma) = \{x \mid x \neq \sigma(x)\}$. Substitutions are homomorphically extended to mappings over $\mathrm{T}(\mathcal{F}, \mathcal{V})$. We write $\sigma(t)$ as $t\sigma$. A substitution σ with $\mathrm{dom}(\sigma) = \{x_1, \dots, x_n\}$ and $\sigma(x_i) = t_i$ is denoted by $\{x_1 := t_1, \dots, x_n := t_n\}$.

We denote an *equation* by $s \approx t$ which is indistinguished from $t \approx s$ and an *inequation* by $s \leqslant t$. We also consider indexed inequations of the form $s \leqslant_i t$ where the index i ranges over $1, \dots, k$. Let $E = \{s_i \circ_i t_i \mid 1 \leq i \leq n, \circ_i \in \{\approx, \leqslant_1, \dots, \leqslant_k\}\}$ be a set of equations and indexed inequations. Then E is said to be *semi-unifiable* if there exists a substitution $\tau, \rho_1, \dots, \rho_k$ such that $\tau(s_i) = \tau(t_i)$

for all $s_i \approx t_i \in E$ and $\rho_j(\tau(s_i)) = \tau(t_i)$ for all $s_i \leqslant_j t_i \in E$; the substitution τ is called a *semi-unifier* of E and the substitutions $\rho_1, \ldots, \rho_k$ are called *residual substitutions* of the semi-unifier τ. A *semi-unification problem* is a problem to ask whether there is a semi-unifier for a given set of equations and (indexed) inequations. A semi-unification problem is said to be *uniform* if $k = 1$ (i.e. the index of inequations is unique); when we think of a uniform semi-unification problem the index of inequations will be omitted. Any (uniform) semi-unification problem E can be reduced to a (uniform) semi-unification problem without equations by replacing $s_i \approx t_i \in E$ with $z_i \leqslant s_i, z_i \leqslant t_i$ using a fresh variable z_i. Thus, one can assume w.l.o.g. that any semi-unification problem deals with only the set of inequations.

Example 1. Let $E = \{\mathsf{f}(\mathsf{h}(y), x) \leqslant \mathsf{f}(x, \mathsf{h}(\mathsf{h}(y)))\}$. Take $\sigma = \{x := \mathsf{h}(y')\}$ and $\rho_1 = \{y := y', y' := \mathsf{h}(y)\}$, where y' is a fresh variable. We have $\mathsf{f}(\mathsf{h}(y), x)\sigma\rho_1 = \mathsf{f}(\mathsf{h}(y), \mathsf{h}(y'))\rho_1 = \mathsf{f}(\mathsf{h}(y'), \mathsf{h}(\mathsf{h}(y))) = \mathsf{f}(x, \mathsf{h}(\mathsf{h}(y)))\sigma$. Thus E is semi-unifiable and σ is a semi-unifier. Note here that $\mathsf{f}(\mathsf{h}(y), x)$ and $\mathsf{f}(x, \mathsf{h}(\mathsf{h}(y)))$ are not unifiable.

3 Symbolic Semi-Unification

In this section, we introduce a notion of symbolic[1] semi-unification. The notion is based on the idea of syntactically representing the substitution ρ of the identity $\rho(\tau(s)) = \tau(t)$ expressing semi-unifiability. This idea goes back to [8]. Our presentation mostly follows a nicer formulation given in [13]. In the literature, various symbols are used as the "place holder" for ρ; we here use ∇ as it is clearly distinguished from substitutions denoted by ρ, τ, σ, etc.

Definition 2 (symbol ∇, ∇-variables, ∇-terms, operator ∇).

1. *We use a* unary special function symbol ∇ *which is supposed to be not contained in $\mathcal{F}$.*
2. *We define* ∇-terms *as follows: (i) $\nabla^i(x)$ where $x \in \mathcal{V}$ and $i \geq 0$ are ∇-terms where $\nabla^i(x)$ abbreviates $\overbrace{\nabla(\cdots\nabla}^{i\text{-times}}(x)\cdots)$; (ii) if $t_1, \ldots, t_n$ are ∇-terms then $f(t_1, \ldots, t_n)$ is a ∇-term for any $f \in \mathcal{F}$ of arity n. Equations of ∇-terms are said to be* ∇-equations.
3. *∇-terms of the form $\nabla^i(x)$ $(i \geq 0)$ are called* ∇-variables. *We denote $\nabla^i(x)$ by x^i. Hence $x^0 = x$, $\nabla(x^i) = x^{i+1}$, and $x^j \trianglelefteq x^i$ for all $j \leq i$. The sets of ∇-variables and ∇-terms are denoted by $\mathcal{V}^*$ and $\mathrm{T}(\mathcal{F}, \mathcal{V}^*)$, respectively. The set of ∇-variables in an object α is denoted by $\mathcal{V}^*(\alpha)$.*
4. *We define a* unary operation ∇ *on ∇-terms recursively as follows: $\nabla(x^i) = x^{i+1}$; $\nabla(f(t_1, \ldots, t_n)) = f(\nabla(t_1), \ldots, \nabla(t_n))$.*

Example 3. A ∇-term $t = \mathsf{f}(x, \mathsf{g}(\nabla(\nabla(y))))$ may be also written as $\mathsf{f}(x, \mathsf{g}(\nabla^2(y)))$ or $\mathsf{f}(x, \mathsf{g}(y^2))$. The set of ∇-variables in t is $\mathcal{V}^*(t) = \{x, y, \nabla(y), \nabla(\nabla(y))\} = \{x, y, y^1, y^2\}$. We have $\nabla(t) = \nabla(\mathsf{f}(x, \mathsf{g}(\nabla^2(y)))) = \mathsf{f}(\nabla(x), \nabla(\mathsf{g}(\nabla^2(y)))) = \mathsf{f}(\nabla(x), \mathsf{g}(\nabla^3(y))) = \mathsf{f}(x^1, \mathsf{g}(y^3))$ and $\nabla^2(t) = \nabla(\nabla(t)) = \mathsf{f}(x^2, \mathsf{g}(y^4))$.

[1] The name "symbolic" is from [19].

Contexts over ∇-terms and the subterm relation on ∇-terms are defined similarly to the usual contexts and subterms. We define ∇-substitutions below.

Definition 4 (∇-substitution).

1. *A* ∇-substitution *is a partial mapping σ from $\mathcal{V}^*$ to $\mathrm{T}(\mathcal{F}, \mathcal{V}^*)$ such that (i) the domain* $\mathrm{dom}(\sigma)$ *of σ is finite; (ii) for each $x \in \mathcal{V}$ there exists at most one i such that $x^i \in \mathrm{dom}(\sigma)$; (iii) for each $x^i, y^j \in \mathrm{dom}(\sigma)$, $y^j \not\trianglelefteq \sigma(x^i)$.*
2. *The application $\sigma(t)$ of a ∇-substitution σ to a ∇-term t is recursively defined as follows: $\sigma(y^j) = y^j$ if $y^i \notin \mathrm{dom}(\sigma)$ for any $i \leq j$; $\sigma(y^j) = \nabla^{j-i}(\sigma(y^i))$ if $y^i \in \mathrm{dom}(\sigma)$ for some $i \leq j$; $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.*
3. *The many-time application $\sigma^*(t)$ of a ∇-substitution σ to a ∇-term t is defined as follows: $\sigma^*(t) = t$ if $x^i \not\trianglelefteq t$ for any $x^i \in \mathrm{dom}(\sigma)$; $\sigma^*(t) = \sigma^*(\sigma(t))$ otherwise.*

We write $\sigma(t)$ as $t\sigma$ and $\sigma^*(t)$ as $t\sigma^*$; in particular, we write $\sigma^*(\nabla(t))$ as $\nabla(t)\sigma^*$. The notion of many-time application of a ∇-substitution to a ∇-term is used to give an invariance of derivation for symbolic semi-unification.

Example 5. The partial mapping $\sigma = \{x := \mathsf{a}, x^1 := \mathsf{b}\}$ is not a ∇-substitution, as the σ does not satisfy the condition (ii). Neither is the partial mapping $\sigma = \{x^1 := \mathsf{f}(y^1), y^1 := \mathsf{b}\}$, as the σ does not satisfy the condition (iii). The partial mapping $\sigma = \{x^1 := y, y^1 := \mathsf{f}(z^2)\}$ is a ∇-substitution, and we have $\sigma(y^3) = \nabla^2(\sigma(y^1)) = \nabla^2(\mathsf{f}(z^2)) = \mathsf{f}(z^4)$ and $\sigma^*(x^2) = \sigma^*(\nabla(y)) = \sigma^*(y^1) = \mathsf{f}(z^2)$.

It may be not so obvious from the definition that $t\sigma^*$ is always well-defined; however, this can be derived from our definition of ∇-substitutions.

Lemma 6. *For any ∇-substitution σ and ∇-term t, $t\sigma^*$ is well-defined.*

Proof. Define, for each ∇-variable x^j, $w(x^j) = \max\{j - i + 1, 0\}$ if $x^i \in \mathrm{dom}(\sigma)$ for some i, and $w(x^j) = 0$ otherwise. Let $\mathcal{W}(t)$ be the multiset of weight of ∇-variables in a ∇-term t. Then if $t \neq t\sigma$ then $\mathcal{W}(t) \gg \mathcal{W}(t\sigma)$, where $\gg$ is the multiset extension (e.g. [1]) of the natural order $>$ on the set of natural numbers. The claim follows from the well-foundedness of $\gg$. $\square$

Lemma 7. *For any ∇-term t and ∇-substitution σ, $\nabla(t)\sigma^* = \nabla(t\sigma^*)\sigma^*$.*

Proof. Let $\mathcal{R} = \{u_g \to v_g \mid u := v \in \sigma\}$ be a TRS (see e.g. [1]), where $()_g$ replaces each variable with a distinct constant. The claim follows from completeness of the TRS $\mathcal{R} \cup \{\nabla(f(x_1, \ldots, x_n)) \to f(\nabla(x_1), \ldots, \nabla(x_n)) \mid f \in \mathcal{F}\}$. $\square$

We now introduce a notion of symbolic semi-unification.

Definition 8 (symbolic semi-unification). *For a set E of ∇-equations, a* semi-unifier *of E is a ∇-substitution σ such that $s\sigma^* = t\sigma^*$ for all $s \approx t \in E$; if E has a semi-unifier, E is said to be* semi-unifiable*. A* symbolic semi-unification problem *asks whether there exists a semi-unifier for a given set of ∇-equations.*

The next lemma is shown using Lemma 7.

Lemma 9 (semi-unifiability is closed under ∇). *Let σ be a ∇-substitution and s, t be ∇-terms. If $s\sigma^* = t\sigma^*$ then $\nabla(s)\sigma^* = \nabla(t)\sigma^*$.*

Remark 10. In contrast to Lemma 9, $s\sigma = t\sigma$ does not necessary imply $\nabla(s)\sigma = \nabla(t)\sigma$: Let $\sigma = \{x^2 := \mathsf{f}(x)\}$, $s = x^3$ and $t = \mathsf{f}(x^1)$. Then $s\sigma = \mathsf{f}(x^1) = t\sigma$, but $\nabla(s)\sigma = x^4\sigma = \mathsf{f}(x^2) \neq \mathsf{f}(\mathsf{f}(x)) = \mathsf{f}(x^2)\sigma = \nabla(t)\sigma$.

The notions of ∇-equality and inconsistency were introduced in [8].

Definition 11 (∇-equality). *For a set E of ∇-equations, the* ∇-equality *generated by E, denoted by $\approx_E$, is the smallest equivalence relation such that (i) $s \approx_E t$ for any $s \approx t \in E$, (ii) $s \approx_E t$ implies $\nabla(s) \approx_E \nabla(t)$, and (iii) for any $f \in \mathcal{F}$, $f(s_1, \ldots, s_n) \approx_E f(t_1, \ldots, t_n)$ iff, for any $i = 1, \ldots, n$, $s_i \approx_E t_i$ holds.*

Definition 12 (inconsistency). *A set E of ∇-equations is* inconsistent *if either (i) $x^i \approx_E s$ with $x^i \trianglelefteq s \notin \mathcal{V}^*$, or (ii) $f(s_1, \ldots, s_m) \approx_E g(t_1, \ldots, t_n)$ with $f \neq g$ for some $f, g \in \mathcal{F}$. Furthermore, E is* consistent *if it is not inconsistent.*

The next lemma will be used heavily in our proof.

Lemma 13. *Let E be a set of ∇-equations. Suppose E is semi-unifiable and let σ be a semi-unifier of E. Then for any ∇-terms u, v, $u \approx_E v$ implies $u\sigma^* = v\sigma^*$.*

Proof. By induction on the derivation of $u \approx_E v$ using Lemma 9. $\square$

4 Symbolic Semi-Unification and Semi-Unification

In this section, we show the equivalence between the consistency and the symbolic semi-unifiability of ∇-equations and the semi-unifiability of the corresponding inequations. Thus, we extend and give a rigorous proof of a result of [8]. A part of the proof will be postponed until Section 6.

We first introduce interpretations of ∇-terms, a key notion of our proof.

Definition 14 (interpretation). *Let τ, ρ be substitutions. An* interpretation *$[\![t]\!]^\tau_\rho \in \mathrm{T}(\mathcal{F}, \mathcal{V})$ of $t \in \mathrm{T}(\mathcal{F}, \mathcal{V}^*)$ is given by $[\![x^i]\!]^\tau_\rho = \rho^i(\tau(x))$; $[\![f(s_1, \ldots, s_n)]\!]^\tau_\rho = f([\![s_1]\!]^\tau_\rho, \ldots, [\![s_n]\!]^\tau_\rho)$.*

Lemma 15. *Let τ, ρ be substitutions. (1) For $t \in \mathrm{T}(\mathcal{F}, \mathcal{V}^*)$, $[\![\nabla(t)]\!]^\tau_\rho = \rho([\![t]\!]^\tau_\rho)$. (2) For $t \in \mathrm{T}(\mathcal{F}, \mathcal{V})$, $[\![t]\!]^\tau_\rho = \tau(t)$.*

Proof. By induction on t. $\square$

A solution $\sigma_\mathsf{u}, \sigma_\mathsf{m}$ of a uniform semi-unification problem can be obtained from a solution σ of symbolic semi-unification problem as follows.

Definition 16 ([8]). *Let σ be a ∇-substitution. Then we define its unification part σ_u and matching part σ_m as below.*

1. *Let $X_0 = \{x \in \mathcal{V} \mid x \in \mathrm{dom}(\sigma)\}$, $X_1 = \{x^i \mid x^i \in \mathrm{dom}(\sigma), i > 0\}$, and $X_2 = (\bigcup\{\mathcal{V}^*(x^i \approx \sigma(x^i)) \mid x^i \in \mathrm{dom}(\sigma)\}) \setminus (\mathcal{V} \cup X_1)$.*

2. *Prepare fresh distinct variables for each $x^i \in X_2$ and let φ be a mapping that assigns to each $x^i \in X_2$ the corresponding fresh variable; let φ be recursively extended as $\varphi(y) = y$; $\varphi(f(t_1, \ldots, t_n)) = f(\varphi(t_1), \ldots, \varphi(t_n))$.*
3. *For each $x^i \in X_1$, put $\sigma_{\mathsf{m}}(x^{i-1}) = \varphi(\sigma(x^i))$ if $i = 1$, and $\sigma_{\mathsf{m}}(\varphi(x^{i-1})) = \varphi(\sigma(x^i))$ otherwise. For each $x^i \in X_2$, put $\sigma_{\mathsf{m}}(x^{i-1}) = \varphi(x^i)$ if $i = 1$, and $\sigma_{\mathsf{m}}(\varphi(x^{i-1})) = \varphi(x^i)$ otherwise.*
4. *For each $x \in X_0$, put $\sigma_{\mathsf{u}}(x) = \varphi(\sigma(x))$.*

Example 17. Let $\sigma = \{x := \mathsf{f}(y, z^1), y^1 := \mathsf{f}(y, z), z^3 := \mathsf{f}(z^1, w^2)\}$. Then $X_0 = \{x\}$, $X_1 = \{y^1, z^3\}$ and $X_2 = \{z^1, z^2, w^1, w^2\}$. Put $\varphi = \{z^1 := z', z^2 := z'', w^1 := w', w^2 := w''\}$. Then we have $\sigma_{\mathsf{m}} = \{y := \mathsf{f}(y, z), z'' := \mathsf{f}(z', w''), z := z', z' := z'', w := w', w' := w''\}$ and $\sigma_{\mathsf{u}} = \{x := \mathsf{f}(y, z')\}$.

Lemma 18. *For any ∇-substitution σ, σ_{u} and σ_{m} are well-defined substitutions.*

Lemma 19. *(1) $\sigma_{\mathsf{m}}^i(x) = \varphi(x^i)$ for $x^i \in X_2$. (2) $\sigma_{\mathsf{m}}^i(x) = \varphi(\sigma(x^i))$ for $x^i \in X_1$.*

Lemma 20. *Let $t \in \mathrm{T}(\mathcal{F}, \mathcal{V}^*)$ such that $\mathcal{V}(t) \cap X_0 = \emptyset$ and $\mathcal{V}^*(t) \setminus \mathcal{V} \subseteq X_2$. Then $[\![t]\!]_{\sigma_{\mathsf{m}}}^{\sigma_{\mathsf{u}}} = \varphi(t)$. In particular, for any $x^i \in X_1$, $[\![\sigma(x^i)]\!]_{\sigma_{\mathsf{m}}}^{\sigma_{\mathsf{u}}} = \varphi(\sigma(x^i))$.*

The next two key lemmas are used to prove the theorem below.

Lemma 21. *For any $t \in \mathrm{T}(\mathcal{F}, \mathcal{V}^*)$ and any ∇-substitution σ, $[\![\sigma^*(t)]\!]_{\sigma_{\mathsf{m}}}^{\sigma_{\mathsf{u}}} = [\![t]\!]_{\sigma_{\mathsf{m}}}^{\sigma_{\mathsf{u}}}$.*

Proof. First prove $[\![\sigma(t)]\!]_{\sigma_{\mathsf{m}}}^{\sigma_{\mathsf{u}}} = [\![t]\!]_{\sigma_{\mathsf{m}}}^{\sigma_{\mathsf{u}}}$ using Lemmas 15, 19 and 20. □

Lemma 22. *Let $s, t \in \mathrm{T}(\mathcal{F}, \mathcal{V}^*)$, and ρ, τ be substitutions. Suppose that $[\![l]\!]_{\rho}^{\tau} = [\![r]\!]_{\rho}^{\tau}$ for any $l \approx r \in E$. Then if $u \approx_E v$ then $[\![u]\!]_{\rho}^{\tau} = [\![v]\!]_{\rho}^{\tau}$.*

Proof. By induction on the derivation of $u \approx_E v$, using Lemma 15. □

Theorem 23 (consistency and semi-unifiability). *For any terms $s, t \in \mathrm{T}(\mathcal{F}, \mathcal{V})$, the following are equivalent: (i) $\{\nabla(s) \approx t\}$ is semi-unifiable, (ii) $\{s \leqslant t\}$ is semi-unifiable, and (iii) $\{\nabla(s) \approx t\}$ is consistent.*

Proof. $(iii) \Rightarrow (i)$ will be shown later (Corollary 45). To show $(i) \Rightarrow (ii)$, use Lemmas 15 and 21. To show $(ii) \Rightarrow (iii)$, use Lemmas 15 and 22. □

$(ii) \Leftrightarrow (iii)$ was obtained in [8]; we incorporate an equivalence with (i).

5 Partial Correctness of Symbolic Semi-Unification

In this section, we give a rule-based symbolic semi-unification procedure and show its partial correctness. Our calculus is a variant of the one given in [8]. Essentially the same calculi are given in [12, 13, 15, 19]. Before giving the procedure, we need a preparation.

Definition 24 (relation $\gtrdot$). *We fix an arbitrary (strict) total order $\gtrdot$ on $\mathcal{V}^*$ satisfying (i) $i > j$ implies $x^i \gtrdot x^j$ and (ii) $x^i \gtrdot y^j$ implies $x^{i+1} \gtrdot y^{j+1}$. The order $\gtrdot$ is extended by $x^i \gtrdot t$ for any $t \notin \mathcal{V}^*$ and $x^i \not\trianglelefteq t$.*

$$
\begin{array}{lll}
\text{Decompose} & \dfrac{\{f(s_1,\ldots,s_n) \approx f(t_1,\ldots,t_n)\} \uplus E}{\{s_1 \approx t_1,\ldots,s_n \approx t_n\} \cup E} & f \in \mathcal{F} \\[2ex]
\text{Reduce} & \dfrac{\{x^i \approx t, C[x^i] \approx u\} \uplus E}{\{x^i \approx t, C[t] \approx u\} \cup E} & x^i \gtrdot t \\[2ex]
\text{Delete} & \dfrac{\{x^i \approx x^i\} \uplus E}{E} & \\[2ex]
\text{Clash} & \dfrac{\{f(s_1,\ldots,s_m) \approx g(t_1,\ldots,t_n)\} \uplus E}{\bot} & f \neq g,\ f,g \in \mathcal{F} \\[2ex]
\text{Check} & \dfrac{\{x^i \approx t\} \uplus E}{\bot} & t \notin \mathcal{V}^*,\ x^i \trianglelefteq t
\end{array}
$$

Fig. 1. Inference rules for symbolic semi-unification

It readily follows from our condition that (i) $i > j$ iff $x^i \gtrdot x^j$ and (ii) $x^i \gtrdot y^j$ iff $x^{i+1} \gtrdot y^{j+1}$. One way to give $\gtrdot$ is to fix some total order $>$ on $\mathcal{V}$ and define $x^i \gtrdot y^j$ iff either $x > y$ or ($x = y$ and $i > j$) as in [8, 15, 19]. But our proof reveals that the abstract condition above is sufficient.

Definition 25 (symbolic semi-unification procedure). *One step derivation using any of inference rules listed in Figure 1 is denoted by $\rightsquigarrow$. Here, the inference rules act on a finite set of ∇-equations and $\uplus$ denotes the disjoint union. For an input of a finite set E_0 of ∇-equations and the relation $\gtrdot$, a symbolic semi-unification procedure non-deterministically constructs a derivation $E_0 \rightsquigarrow E_1 \rightsquigarrow \cdots$ (possibly following some fixed derivation strategy). The derivation may be finite or infinite, and it is* maximal *if it does not end with E_k for which a further application of an inference rule is possible. A symbolic semi-unification procedure (following a fixed derivation strategy)* terminates *if any derivation (following that derivation strategy) is finite.*

The reflexive transitive closure of $\rightsquigarrow$ is denoted by $\overset{*}{\rightsquigarrow}$.

Example 26. Let the total order $\gtrdot$ be given by $w^i \gtrdot x^j \gtrdot y^k \gtrdot z^l$ for any i, j, k, l. Consider $E = \{y^3 \approx z, w^3 \approx x, x^2 \approx \mathsf{f}^2(y), x^1 \approx \mathsf{f}(w^2)\}$. Then we have

$$
\begin{array}{l}
\{y^3 \approx z, w^3 \approx x, x^2 \approx \mathsf{f}^2(y), x^1 \approx \mathsf{f}(w^2)\} \\
\quad \rightsquigarrow \{y^3 \approx z, w^3 \approx x, \underline{\mathsf{f}(w^3)} \approx \mathsf{f}^2(y), x^1 \approx \mathsf{f}(w^2)\} \\
\quad \rightsquigarrow \{y^3 \approx z, w^3 \approx x, \underline{w^3 \approx \mathsf{f}(y)}, x^1 \approx \mathsf{f}(w^2)\} \\
\quad \rightsquigarrow \{y^3 \approx z, \underline{\mathsf{f}(y)} \approx x, w^3 \approx \mathsf{f}(y), x^1 \approx \mathsf{f}(w^2)\} \\
\quad \rightsquigarrow \{y^3 \approx z, x \approx \mathsf{f}(y), w^3 \approx \mathsf{f}(y), \underline{\mathsf{f}(y^1)} \approx \mathsf{f}(w^2)\} \\
\quad \rightsquigarrow \{y^3 \approx z, x \approx \mathsf{f}(y), w^3 \approx \mathsf{f}(y), \underline{w^2 \approx y^1}\} \\
\quad \rightsquigarrow \{y^3 \approx z, x \approx \mathsf{f}(y), \underline{y^2} \approx \mathsf{f}(y), w^2 \approx y^1\} \\
\quad \rightsquigarrow \{z \approx \underline{\mathsf{f}(y^1)}, x \approx \mathsf{f}(y), y^2 \approx \mathsf{f}(y), w^2 \approx y^1\}.
\end{array}
$$

Here, modified parts are underlined.

Remark 27. In [8], the following (almost) rule-based procedure is given: (1) Apply Decompose as many times as possible; Apply Clash or Check if possible. (2) Fix a total order $>$ on variables. Consider a ground TRS $R = \{l \to r \mid l \approx r \in E, l > r\}$. Then: "For each rule, reduce each side (if reducible) by a single step of rewriting by other rules. If no rule can be rewritten any further, report SUCCESS. Otherwise replace the rule by the new equation thus obtained and go to Step (1)."

It is claimed in [8] that this procedure is terminating. The description "For each rule, ..." is difficult to interpret: for example,

$$\begin{aligned}\{\mathsf{h}(\mathsf{g}(y^1), \mathsf{g}(z^1), \mathsf{g}(x^1)) \approx \mathsf{h}(x, y, z)\} &\rightsquigarrow \{x \approx \underline{\mathsf{g}(y^1)}, y \approx \underline{\mathsf{g}(z^1)}, z \approx \underline{\mathsf{g}(x^1)}\} \\ &\rightsquigarrow \{x \approx \underline{\mathsf{g}^2(z^2)}, y \approx \underline{\mathsf{g}^2(x^2)}, z \approx \underline{\mathsf{g}^2(y^2)}\} \\ &\rightsquigarrow \{x \approx \underline{\mathsf{g}^4(y^4)}, y \approx \underline{\mathsf{g}^4(z^4))}, z \approx \underline{\mathsf{g}^4(x^4)}\} \\ &\rightsquigarrow \cdots\end{aligned}$$

should not be the case, as they claim that their procedure terminates.

Remark 28. Recall the following inference rules of the unification procedure corresponding to Reduce and Check:

$$\text{Reduce}' \quad \frac{\langle \{x \approx t\} \uplus E, \sigma\rangle}{\langle \{x := t\}(E), \{x := t\} \circ \sigma\rangle}\; x \notin \mathcal{V}(t) \qquad \text{Check}' \quad \frac{\langle \{x \approx s\} \uplus E, \sigma\rangle}{\bot}\; x \trianglelefteq s, s \notin \mathcal{V}$$

Two differences can be observed:

- the equations part and the substitution part are separated, and
- Reduce$'$ uses the substitution (replacing all occurrences of x), while Reduce uses the replacement (replacing a single occurrence of x^i).

In the unification procedure, the substitution part can be naturally separated from the equations part, as the substitution $\{x := t\}$ is not needed again in the equation part. But in the case of semi-unification, this is not the case:

$$\begin{array}{ll} \{x^1 \approx \mathsf{f}(x, y^2), y^1 \approx \mathsf{g}(x), y^3 \approx \mathsf{g}(z)\} & \{x^1, x, y^2, y^1, x, y^3, z\} \\ \rightsquigarrow \{x^1 \approx \mathsf{f}(x, y^2), y^1 \approx \mathsf{g}(x), \underline{\mathsf{g}(x^2)} \approx \mathsf{g}(z)\} & \{x^1, x, y^2, y^1, x, x^2, z\} \\ \rightsquigarrow \{x^1 \approx \mathsf{f}(x, y^2), y^1 \approx \mathsf{g}(x), \overline{x^2 \approx z}\} & \{x^1, x, y^2, y^1, x, x^2, z\} \\ \rightsquigarrow \{x^1 \approx \mathsf{f}(x, y^2), y^1 \approx \mathsf{g}(x), \underline{\mathsf{f}(x^1, y^3)} \approx z\} & \{x^1, x, y^2, y^1, x, x^1, y^3, z\} \\ \rightsquigarrow \cdots & \end{array}$$

At the first line, $x^1 := \mathsf{f}(x, y^2)$ can not be applied to other equations, while it can be applied to the equation $x^2 \approx z$ at the third line. Thus, a solved equation needs to be kept in the system for the future simplifications or for the case the equation itself is simplified in the future. Hence, it is not possible to split off the solved equations as the substitution part. Furthermore, since the substitution does not eliminate the future need of the application of the same substitution, it seems natural to use the replacement in Reduce instead of the substitution.

It is also observed that the derivation from the first line to the fourth line strictly increases the multiset of ∇-variables in the equation, which is listed at the right. Hence it is difficult to adapt a termination proof similar to the one applied to calculi for unification (see e.g. [1]).

Example 29. In fact, our calculus is not terminating—this is witnessed by the following derivation.

$$\begin{array}{l}
\{x \approx z, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\underline{\mathsf{g}(y)} \approx z, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\mathsf{g}(y) \approx \underline{\mathsf{g}(x)}, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\underline{y \approx x}, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\underline{\mathsf{g}(z)} \approx x, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\mathsf{g}(z) \approx \underline{\mathsf{g}(y)}, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\underline{z \approx y}, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\underline{\mathsf{g}(x)} \approx y, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\mathsf{g}(x) \approx \underline{\mathsf{g}(z)}, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \{\underline{x \approx z}, x \approx \mathsf{g}(y), y \approx \mathsf{g}(z), z \approx \mathsf{g}(x)\} \\
\quad \rightsquigarrow \cdots
\end{array}$$

We note that because of some postponed applications of Reduce, Check is not applicable in the derivation. This infinite derivation is also valid in the calculus given in [15]. In Section 6, we will give a sufficient criterion on derivation strategies under which any derivation terminates.

Remark 30. The infinite derivation above is not possible, if one adopts a variant of Reduce using substitution (instead of the replacement):

$$\text{Reduce}'' \quad \frac{\{x^i \approx t\} \uplus E}{\{x^i \approx t\} \cup \{x^i := t\}(E)} \; x^i \gtrdot t$$

Rule-based semi-unification calculi in [8, 15] use the replacement, and those in [12, 13, 19] use the substitution. We note that any substitution can be simulated by repeated applications of replacement. We refer Lemma 43 for the termination of the calculus obtained by replacing the Reduce by Reduce″—termination of such calculus under a particular derivation strategy is also obtained in [12].

Remark 31. Another difference of the rule-based semi-unification calculi in the literature is whether the transformation $\nabla(f(t_1, \ldots, t_n)) = f(\nabla(t_1), \ldots, \nabla(t_n))$ is admitted in the course of derivations. The calculi in [15, 19] admit such flexibility. Adding such flexibility, however, causes another non-terminating derivation[2].

We now give several properties of finite derivations.

Lemma 32. *Suppose $E \overset{*}{\rightsquigarrow} E'$ with $E' \neq \bot$. Then $\approx_E = \approx_{E'}$.*

Using Lemma 13, we have

Corollary 33. *If $E \overset{*}{\rightsquigarrow} E' \neq \bot$, then E is semi-unifiable iff E' is semi-unifiable.*

Using Lemma 13 and Corollary 33, partial correctness of our symbolic semi-unification procedure is obtained.

Theorem 34 (partial correctness). *Let E be a finite set of ∇-equations. (1) If $E \overset{*}{\rightsquigarrow} \bot$ then E is not semi-unifiable. (2) If $E \overset{*}{\rightsquigarrow} E' \neq \bot$ and no inference rules are applicable to E', then E is semi-unifiable.*

[2] The authors learned this observation from an anonymous reviewer.

6 Termination of Symbolic Semi-Unification Procedure

In this section, we show termination of our rule-based symbolic semi-unification procedure under an assumption on the derivation strategy employed.

For the proof, we introduce a new relation.

Definition 35 (variable relation). *A* relation $>_v$ *on* ∇*-variables, called the* variable relation consistent with a set of ∇-equations E and the order $\gtrdot$, *is the smallest transitive relation satisfying the following conditions: (i) if* $x^i \approx_E y^j$ *with* $x^i \gtrdot y^j$ *then* $x^i >_v y^j$*, (ii) if* $x^i \approx_E C[y^j]$ *with* $C[y^j] \notin \mathcal{V}^*$ *for some strict context* C *then* $x^i >_v y^j$*. Here, a context* C *is* strict *if* C *is not of the form* $C'[\nabla(\Box)]$ *for some context* C'*. The reflexive closure of* $>_v$ *is denoted by* $\geq_v$.

Lemma 36. *Let* $>_v$ *be a variable relation consistent with* E *and* $\gtrdot$*. If* $x^i >_v y^j$ *then* $x^i \approx_E C[y^j]$ *for some context* C*; furthermore, if* $C = \Box$ *then* $x^i \gtrdot y^j$.

Lemma 37. *Let* X *be a finite set of variables. Let* $a_0, a_1, \ldots$ *be an infinite sequence of* ∇*-variables from* $X^* = \{x^i \mid x \in X, i \geq 0\}$*. Then there exist indexes* i, j *with* $i < j$ *such that* $a_j = \nabla^k(a_i)$ *for some* $k \geq 0$.

Definition 38 ($\mathcal{M}(t)$, $\mathcal{M}(E)$). *The multiset* $\mathcal{M}(t)$ *of* ∇*-variables that occur* maximally *in a* ∇*-term* t *is defined like this:* $\mathcal{M}(x^i) = \{x^i\}$*;* $\mathcal{M}(f(t_1, \ldots, t_n)) = \biguplus_i \mathcal{M}(t_i)$*. For a finite set* E *of* ∇*-equations, we put* $\mathcal{M}(E) = \biguplus\{\mathcal{M}(s) \uplus \mathcal{M}(t) \mid s \approx t \in E\}$*. Here,* $\uplus$ *denotes the multiset union.*

The next property is well-known (see e.g. [1]).

Proposition 39. *Let* $\gg_v$ *be the multiset extension of* $>_v$ *and* $\geqq_v$ *its reflexive closure. Let* $M_0 \geqq_v M_1 \geqq_v \cdots$ *be an infinite sequence of finite multisets such that* $M_i \gg_v M_{i+1}$ *for infinitely many indexes* i*. Then there is an infinite sequence* $a_0 \geq_v a_1 \geq_v \cdots$ *with* $a_i \in M_i$ *such that* $a_i >_v a_{i+1}$ *for infinitely many indexes* i.

Lemma 40. *Let* E_0 *be a finite set of* ∇*-equations,* $>_v$ *the variable relation consistent with* E_0 *and* $\gtrdot$*. If* $E_0 \overset{*}{\rightsquigarrow} E \rightsquigarrow E' \neq \bot$ *then* $\mathcal{M}(E) \geqq_v \mathcal{M}(E')$*. In particular, if* $E \rightsquigarrow E'$ *is by Reduce, then* $\mathcal{M}(E) \gg_v \mathcal{M}(E')$.

Proof. Distinguish the cases by the inference rule used in $E \rightsquigarrow E'$. □

Theorem 41 (termination of symbolic semi-unification procedure). *Every derivation starting from a consistent finite set of* ∇*-equations is finite.*

Proof. Suppose $E_0 \rightsquigarrow E_1 \rightsquigarrow \cdots$ be an infinite derivation and E_0 is consistent. Then, Clash and Check can not be used in this derivation. Decompose and Delete do not increase $\mathcal{M}(E_i)$ but reduce the number of symbols. Hence there does not exists an index j such that the Decompose and Delete are used for all $E_i \rightsquigarrow E_{i+1}$ with $i > j$. Thus there are infinitely many i such that Reduce is used on $E_i \rightsquigarrow E_{i+1}$. Hence $\mathcal{M}(E_0) \geqq_v \mathcal{M}(E_1) \geqq_v \cdots$ and there are infinitely many i such that $\mathcal{M}(E_i) \gg_v \mathcal{M}(E_{i+1})$ by Lemma 40. Then, by Proposition 39, we have an infinite sequence $x_0^{i_0} >_v x_1^{i_1} >_v \cdots$. Thus by Lemma 37 there exists indexes

k, l ($k < l$) such that $i_k \leq i_l$ and $x_k = x_l = x$. Since $>_v$ is transitive relation, $x_k^{i_k} >_v x_l^{i_l}$. Hence by Lemma 36, $x^{i_k} \approx_E C[x^{i_l}]$ for some context C such that $C = \Box$ implies $x^{i_k} \gtrdot x^{i_l}$. $C \neq \Box$ contradicts with the consistency of E. If $C = \Box$ then $x^{i_k} \gtrdot x^{i_l}$ contradicts $i_k \leq i_l$. Thus there exists no infinite derivation. $\Box$

The above theorem motivates the following definition.

Definition 42 (refutational completeness). *A derivation strategy is said to be* refutationally complete *if any maximal derivation starting from an inconsistent set of ∇-equations and following that strategy is finite and ends with $\perp$.*

The next lemma gives a concrete example of refutationally complete derivation strategy. Note that the rule Reduce$''$ is given in Remark 30.

Lemma 43. *A derivation strategy subject to using Reduce$''$ in place of Reduce and applying Check whenever possible is refutationally complete.*

Proof. Suppose $E_0 \rightsquigarrow E_1 \rightsquigarrow \cdots$ be a maximal derivation and E_0 is inconsistent. If Clash or Check is used then we are done. Otherwise, from the proof of Theorem 41, we have a sequence of ∇-variables $x^{i_k} = x_k^{i_k} >_v x_{k+1}^{i_{k+1}} >_v \cdots >_v x_l^{i_l} = x^{i_l}$ with $i_k \leq i_l$. This means that there exist $x^{i_k} \approx C_k[x_{k+1}^{i_{k+1}}] \in E_{i_k}, \ldots, x_{l-1}^{i_{l-1}} \approx C_l[x^{i_l}] \in E_{i_{l-1}}$. Since Reduce$''$ is simulated in our derivation, every $x_{j+1}^{i_{j+1}}$ in $C_j[x_{j+1}^{i_{j+1}}]$ is replaced by $C_{j+1}[x_{j+2}^{i_{j+2}}]$ for $j = k+1, \ldots, l-1$. Hence $x^{i_k} \approx C_k[x_{k+1}^{i_{k+1}}] \in E_{i_k}$ has the descendant $x^{i_k} \approx C[x^{i_l}] \in E_{i_l}$ such that $C = \Box$ implies $x^{i_k} \gtrdot x^{i_l}$. If $C \neq \Box$ then Check can be applied, which contradicts our assumption. If $C = \Box$ then $x^{i_k} \gtrdot x^{i_l}$ contradicts $i_k \leq i_l$. $\Box$

The next theorem immediately follows from Theorems 34 and 41.

Theorem 44 (total correctness). *The symbolic semi-unification procedure terminates if it follows a refutationally complete derivation strategy; either the input E is semi-unifiable and any maximal derivation ends with a set of ∇-equations or E is not semi-unifiable and any maximal derivation ends with $\perp$.*

We now obtain Theorem 4.1 of [8], on which their correctness proof is based, as a corollary.

Corollary 45 (consistency and semi-unifiability [8]). *Let E be a finite set of ∇-equations. Then E is consistent iff E is semi-unifiable.*

Proof. ($\Rightarrow$) Use Theorem 44 and Lemma 32. ($\Leftarrow$) Use Lemma 13. $\Box$

7 Conclusion

We have revisited rule-based calculi for uniform semi-unification, on which efficient uniform semi-unification procedures [8, 15] are based. We have given a new characterization of symbolic semi-unification and extended the correspondence between symbolic semi-unifiability and uniform semi-unifiability. For a rule-based calculus of symbolic semi-unification, which is given in a general form essentially including those of [8, 12, 15, 19], we have shown its termination and correctness under refutationally complete derivation strategy.

Acknowledgment. Thanks are due to anonymous referees for many valuable comments. The authors are grateful for Yoshihito Toyama for helpful comments. This work was partially supported by a grant from JSPS No. 23500002.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
2. Dörre, J., Rounds, W.C.: On subsumption and semiunification in feature algebras. In: Proc. of LICS 1990, pp. 300–310. IEEE (1990)
3. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and Disproving Termination of Higher-Order Functions. In: Gramlich, B. (ed.) FroCoS 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005)
4. Henglein, F.: Algebraic properties of semi-unification. In: Proc. of LFP 1988. ACM Press (1988)
5. Henglein, F.: Polymorphic Type Inference and Semi-Unification. Ph.D. thesis, Rutgers University (1989)
6. Henglein, F.: Fast Left-Linear Semi-Unification. In: Akl, S.G., Koczkodaj, W.W., Fiala, F. (eds.) ICCI 1990. LNCS, vol. 468, pp. 82–91. Springer, Heidelberg (1991)
7. Henglein, F.: Type inference with polymorphic recursion. TOPLAS 15(2), 253–289 (1993)
8. Kapur, D., Musser, D., Narendran, P., Stillman, J.: Semi-unification. TCS 81(2), 169–187 (1991)
9. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: Computational consequences and partial solutions of a generalized unification problem. In: Proc. of LICS 1989, pp. 98–104. IEEE (1989)
10. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: The undecidability of the semi-unification problem. IC 102(1), 83–101 (1993)
11. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: An analysis of ML typability. JACM 41(2), 368–398 (1994)
12. Leiß, H.: Polymorphic Recursion and Semi-Unification. In: Börger, E., Kleine Büning, H., Richter, M.M. (eds.) CSL 1989. LNCS, vol. 440, pp. 211–224. Springer, Heidelberg (1990)
13. Leiß, H., Henglein, F.: A Decidable Case of the Semi-Unification Problem. In: Tarlecki, A. (ed.) MFCS 1991. LNCS, vol. 520, pp. 318–327. Springer, Heidelberg (1991)
14. Lushman, B., Cormack, G.V.: A larger decidable semiunification problem. In: Proc. of PPDP 2007, pp. 143–152. ACM Press (2007)
15. Oliart, A., Snyder, W.: Fast algorithms for uniform semi-unification. JSC 37(4), 455–484 (2004)
16. Payet, É.: Loop detection in term rewriting using the eliminating unfoldings. TCS 403(2-3), 307–327 (2008)
17. Pudlák, P.: On a unification problem related to Kreisel's conjecture. Commentationes Mathematicae Universitatis Carolinae 29(3), 551–556 (1988)
18. Purdom Jr., P.W.: Detecting Looping Simplifications. In: Lescanne, P. (ed.) RTA 1987. LNCS, vol. 256, pp. 54–61. Springer, Heidelberg (1987)
19. Ružička, P.: An Efficient Decision Algorithm for the Uniform Semi-Unification Problem. In: Tarlecki, A. (ed.) MFCS 1991. LNCS, vol. 520, pp. 415–425. Springer, Heidelberg (1991)

Deciding WQO for Factorial Languages

Aistis Atminas[1], Vadim Lozin[1,⋆], and Mikhail Moshkov[2]

[1] DIMAP and Mathematics Institute
University of Warwick, Coventry, CV4 7AL, UK
{A.Atminas,V.Lozin}@warwick.ac.uk
[2] CEMSE Division, King Abdullah University of Science and Technology
Thuwal 23955-6900, Saudi Arabia
mikhail.moshkov@kaust.edu.sa

Abstract. A language is factorial if it is closed under taking factors (i.e. contiguous subwords). Every factorial language can be described by an antidictionary, i.e. a minimal set of forbidden factors. We show that the problem of deciding whether a factorial language given by a *finite* antidictionary is well-quasi-ordered under the factor containment relation can be solved in polynomial time.

Keywords: well-quasi-ordering, factorial language, polynomial-time algorithm.

1 Introduction

Well-quasi-ordering (wqo) is a highly desirable property and frequently discovered concept in mathematics and theoretical computer science [6,9]. One of the most remarkable recent results in this area is the proof of Wagner's conjecture stating that the set of all finite graphs is well-quasi-ordered by the minor relation [11]. However, the subgraph or induced subgraph relation is not a well-quasi-order. Other examples of important relations that are not well-quasi-orders are pattern containment relation on permutations [12], embeddability relation on tournaments [3], minor ordering of matroids [7], factor (contiguous subword) relation on words [10]. On the other hand, each of these relations may become a well-quasi-order under some additional restrictions. In the present paper, we study restrictions given in the form of obstructions, i.e. minimal excluded ("forbidden") elements (precise definitions and examples will be given in the next section). The fundamental problem of our interest is the following: given a partial order P and a *finite* set of obstructions Z, determine if the set of elements of P containing no elements from Z forms a well-quasi-order. This problem was studied for the induced subgraph relation on graphs [8], the pattern containment relation on permutations [2], the embeddability relation on tournaments [3], the minor ordering of matroids [7]. However, the decidability of this problem

⋆ The author gratefully acknowledges support from DIMAP - the Center for Discrete Mathematics and its Applications at the University of Warwick, and from EPSRC, grant EP/I01795X/1.

has been shown only for one or two forbidden elements (graphs, permutations, tournaments, matroids). Whether this problem is decidable for larger numbers of forbidden elements is an open question. In the present paper, we answer this question positively for factorial languages.

All preliminary information related to the topic of the paper can be found in Section 2.

2 Preliminaries

2.1 Partial Orders and WQO

For a set A we denote by A^2 the set of all ordered pairs of (not necessarily distinct) elements from A. A *binary relation* on A is a subset of A^2. If a binary relation $\mathcal{R} \subset A^2$ is

- *reflexive*, i.e. $(a, a) \in \mathcal{R}$ for each $a \in A$,
- *transitive*, i.e. $(a, b) \in \mathcal{R}$ and $(b, c) \in \mathcal{R}$ imply $(a, c) \in \mathcal{R}$,

then $\mathcal{R}$ is a *quasi-order* (also known as *pre-order*). If additionally $\mathcal{R}$ is

- *antisymmetric*, i.e. $(a, b) \in \mathcal{R}$ and $(b, a) \in \mathcal{R}$ imply $a = b$,

then $\mathcal{R}$ is a *partial order.*

We say that two elements $a, b \in A$ are comparable with respect to $\mathcal{R}$ if either $(a, b) \in \mathcal{R}$ or $(b, a) \in \mathcal{R}$. A set of pairwise comparable elements of A is called a *chain* and a set of pairwise incomparable elements of A is called an *antichain.*

A quasi-ordered set is *well-quasi-ordered* if it contains

- neither infinite strictly decreasing chains, in which case we say that the set is *well-founded*,
- nor infinite antichains.

All examples of quasi-orders in this paper will be antisymmetric (i.e. partial orders) and well-founded, in which case well-quasi-orderability is equivalent to the non-existence of infinite antichains.

Examples.

(1) Let A be the set of all finite simple (i.e. undirected, without loops and multiple edges) graphs. If a graph $H \in A$ can be obtained from a graph $G \in A$ by a (possibly empty) sequence of
    - vertex deletions, then H is an induced subgraph of G,
    - vertex deletions and edge deletions, then H is a subgraph of G,
    - vertex deletions, edge deletions and edge contractions, then H is a minor of G,
    - vertex deletions and edge contractions, then H is an induced minor of G.

According to the celebrated result of Robertson and Seymour [11] the minor relation on the set of graphs is a well-quasi-order. However, this is not the case for the subgraph, induced subgraph and induced minor relation. Indeed, it is not difficult to see that the set of all chordless cycles $C_3, C_4, C_5, \ldots$ creates an infinite antichain with respect to both subgraph and induced subgraph relations, and the complements of the cycles form an infinite antichain with respect to the induced minor relation. Besides, the set of so-called H-graphs (i.e. graphs represented in Figure 1) also forms an infinite antichain with respect to subgraph and induced subgraph relations.

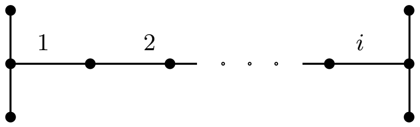


Fig. 1. The graph H_i

(2) Let A be the set of all finite permutations. We say that a permutation $\pi \in A$ of n elements is contained in a permutation $\rho \in A$ of k elements $(n \leq k)$ as a *pattern*, if π can be obtained from ρ by deleting some (possibly none) elements and renaming the remaining elements consecutively in the increasing order. Obviously, the pattern containment relation is a well-founded partial order. However, whether it is a quasi-order is not an obvious fact. Finding an infinite antichain of permutations becomes much easier if we associate to each permutation its permutation graph. Let π be a permutation on the set $N = \{1, 2, \ldots, n\}$. The permutation graph G_π of π is the graph with vertex set N in which two vertices i and j are adjacent if and only if they form an inversion in π (i.e. $i < j$ and $\pi(i) > \pi(j)$). It is not difficult to see that if ρ contains π as a pattern, then G_ρ contains G_π as an induced subgraph. Therefore, if $G_1, G_2, \ldots$ is an infinite antichain of permutation graphs with respect to the induced subgraph relation, then the corresponding permutations form an infinite antichain with respect to the pattern containment relation. Since the H-graphs (Figure 1) are permutation graphs (which is easy to see), we conclude that the pattern containment relation on permutations is not a well-quasi-order.

(3) Let A be the set of all finite words in a finite alphabet. A word $\alpha \in A$ is said to be a *factor* of a word $\beta \in A$ if α can be obtained from β by omitting a (possibly empty) suffix and prefix. If the alphabet contains at least two symbols, say 1 and 0, the factor containment relation is not a well-quasi-order, since it necessarily contains an infinite antichain, for instance, 010, 0110, 01110, etc.

2.2 Hereditary Properties of Partial Orders

Let $(A, \mathcal{R})$ be a well-founded partial order. A property on A is a subset of A. A property $P \subseteq A$ is *hereditary* (with respect to $\mathcal{R}$) if $x \in P$ implies $y \in P$ for every $y \in A$ such that $(y, x) \in \mathcal{R}$. Hereditary properties are also known as lower ideals or downward closed sets.

Examples.

- If A is the set of all finite graphs and $\mathcal{R}$ is the minor relation, then a hereditary property on A is known as a *minor-closed* class of graphs.
- If A is the set of all finite graphs and $\mathcal{R}$ is the subgraph relation, then a hereditary property on A is known as a *monotone* class of graphs.
- If A is the set of all permutations and $\mathcal{R}$ is the pattern containment relation, then a hereditary property on A is known as a *pattern* class or *pattern avoiding* class.
- If A is the set of all words in a finite alphabet and $\mathcal{R}$ is the factor containment relation, then a hereditary property on A is known as a *factorial language.*

The word "avoiding" used in the terminology of permutations suggests that a hereditary property can be described in terms of "forbidden" elements. To better explain this idea, let us introduce the following notation: given a set $Z \subseteq A$, we denote

$$Free(Z) := \{a \in A \mid (z, a) \notin \mathcal{R} \ \forall z \in Z\}.$$

Obviously, for any $Z \subseteq A$, the set $Free(Z)$ is hereditary. On the other hand, for any hereditary property $P \subseteq A$ there is a unique minimal set $Z \subseteq A$ such that $P = Free(Z)$. We call Z the set of *forbidden elements* for $Free(Z)$ and observe that a minimal set of forbidden elements is necessarily an antichain.

Examples.

- Since the minor relation on graphs contains no infinite antichains, any minor-closed class of graphs can be described by a *finite* set of forbidden minors. In particular, for the class of planar graphs the set of minimal forbidden minors consists of K_5, the complete graph on 5 vertices, and $K_{3,3}$, the complete bipartite graph with 3 vertices in each part.
- The set of minimal forbidden permutations for a pattern avoiding class is also known as the *base* of the class.
- The set of minimal forbidden words for a factorial language is also known as the *antidictionary* of the language.

In the above notation, the problem of our interest can be stated as follows:

Problem 1. Given a finite set $Z \subset A$, determine if $Free(Z)$ is well-quasi-ordered with respect to $\mathcal{R}$.

This question is not applicable to the minor relation on graphs, since this relations is a well-quasi-order. For hereditary properties of graphs with respect to the subgraph relation, Problem 1 has a simple solution which is due to Ding [5]:

a monotone class of graphs is well-quasi-ordered by the subgraph relation if and only if it contains finitely many cycles and finitely many H-graphs. Therefore, if Z is finite, then $Free(Z)$ is well-quasi-ordered with respect to the subgraph relation if and only if Z includes a chordless path (or an induced subgraph of a chordless path), because otherwise $Free(Z)$ contains infinitely many cycles.

For other relations, such as the induced subgraph relation on graphs or pattern containment relation on permutations, only partial results are available, where Z contains one or two elements (see e.g. [1,8]). Whether this problem is decidable for larger numbers of forbidden elements is an open question. In the present paper, we study Problem 1 for factorial languages and show that the problem is efficiently solvable for any finite set Z. To this end, we use the result from [4] which allows representing a factorial language defined by a finite antidictionary in the form of a finite deterministic automaton.

2.3 Languages and Automata

Let $k \geq 2$ be a natural number and $E_k = \{0, 1, \dots, k-1\}$ be an alphabet. A *finite deterministic automaton* over E_k is a triple $A = (G, q_0, Q)$, where

– G is a finite directed graph, possibly with multiple edges and loops, in which the edges are labeled with letters from E_k in such a way that any two edges leaving the same node have different labels,
– q_0 is a node of G, called the *start node*, and
– Q is a nonempty set of nodes of G, called the *terminal nodes.*

A directed path in G is any sequence $v_1, e_1, \dots, v_m, e_m, v_{m+1}$ of nodes v_i and edges e_j such that for each $j = 1, \dots, m$, the edge e_j is directed from v_j to v_{j+1}. We emphasize that both nodes and edges can appear in such a path repeatedly.

With each directed path τ in the graph G we associate a word over E_k by reading the labels of the edges of τ (listed along the path) and denote this word by $w(\tau)$. A directed path in G will be called an A-path if it starts at the node q_0 and ends at a terminal node.

Let α be a word over E_k. We say that an automaton $A = (G, q_0, Q)$ accepts α if there is an A-path τ such that $w(\tau) = \alpha$. The set of all words accepted by A is called the language accepted (or recognized) by A and this language is denoted $L(A)$. It is well-known that the set of languages accepted by finite deterministic automata are precisely the regular languages.

The following result was proved in [4].

Theorem 2. *Given a set Z of n words over E_k, in time $O(nk)$ one can construct a finite deterministic automaton A such that $L(A)$ coincides with the factorial language $Free(Z)$.*

We call an automaton $A = (G, q_0, Q)$ *reduced* if for each node of G there exists an A-path containing this node. It is not difficult to see that any finite automaton can be transformed into an equivalent (i.e. accepting the same language) reduced automaton in polynomial time. This observation together with Theorem 2 reduce Problem 1 to the following one:

Problem 3. Given a finite deterministic automaton A, determine if $L(A)$ is well-quasi-ordered with respect to the factor containment relation.

In the next two sections, we give an efficient solution to this problem.

3 Auxiliary Results

Given a word α, we denote by $|\alpha|$ the length of α, i.e. the number of letters in α. Also, α^i denotes concatenation of i copies of α and is called the i-th power of α.

A word $\alpha = \alpha_1 \dots \alpha_n$ is called a *periodic word with period p* if

- either $p \geq n$
- or $p < n$ and $\alpha_i = \alpha_{i+p}$ for $i = 1, \dots, n-p$.

A word γ will be called a *left extension of a power of a word α* if γ can be represented in the form $\sigma\alpha^i$, where σ is a suffix of α. Similarly, γ will be called a *right extension of a power* of α if γ can be represented in the form $\alpha^i\sigma$, where σ is a prefix of α. Directly from the definition we obtain the following conclusion.

Lemma 4. *A word γ is a left extension of a power of α if and only if the word $\gamma\alpha$ is a periodic word with period $|\alpha|$. A word γ is a right extension of a power of α if and only if the word $\alpha\gamma$ is a periodic word with period $|\alpha|$.*

Now we prove a number of further auxiliary results.

Lemma 5. *Let γ, α, δ be words such that either γ is a left extension of a power of α or δ is a right extension of a power of α. Then the set $\{\gamma\alpha^i\delta : i = 0, 1, 2, \dots\}$ is a chain, i.e. any two words in this set are comparable.*

Proof. Let $\gamma\alpha^i\delta$ and $\gamma\alpha^j\delta$ be two words with $i < j$. If γ is a left extension of a power of α, then the word γ can be represented as $\sigma\alpha^k$ where σ is a suffix of α. Therefore, the word $\gamma\alpha^i\delta$ can be obtained from the word $\gamma\alpha^j\delta$ by removing a prefix of length $(j-i)|\alpha|$. Similarly, if δ is a right extension of a power of α, then the word $\gamma\alpha^i\delta$ can be obtained from the word $\gamma\alpha^j\delta$ by removing a suffix of length $(j-i)|\alpha|$. □

Lemma 6. *Let γ, α, δ be words such that γ is not a left extension of a power of α, and δ is not a right extension of a power of α. Then the set of words $\{\gamma\alpha^i\delta : i = 1, 2, \dots\}$ contains an infinite antichain.*

Proof. Let p be a natural number such that $p\,|\alpha| > |\gamma| + |\delta|$. We will show that the set of words $\{\gamma\alpha^{ip}\delta : i = 1, 2, \dots\}$ is an antichain. Assume the contrary, i.e. assume there are numbers $0 < i < j$ such that the words $\gamma\alpha^{ip}\delta$ and $\gamma\alpha^{jp}\delta$ are comparable, i.e. one of them is a factor of the other. We know that $\left|\gamma\alpha^{ip}\delta\right| < \left|\alpha^{jp}\right|$. Therefore, either $\gamma\alpha^{ip}$ is a factor of α^{jp}, or $\alpha^{ip}\delta$ is a factor of α^{jp}. In the first case, $\gamma\alpha$ is a periodic word with period $|\alpha|$. In the second case, $\alpha\delta$ is a periodic word with period $|\alpha|$. Both cases are impossible according to Lemma 4, since neither γ is a left extension of a power of α nor δ is a right extension of a power of α. □

Lemma 7. *Let $\gamma, \alpha, \delta, \beta$ be words. If δ is not a right extension of a power of α, then the word $\gamma\alpha^{|\beta|}\delta$ is not a left extension of a power of β.*

Proof. Let us assume the contrary, i.e. assume that $\gamma\alpha^{|\beta|}\delta$ is a left extension of a power of β. Then, by Lemma 4, the word $\varepsilon = \gamma\alpha^{|\beta|}\delta\beta$ is a periodic word with period $|\beta|$ and, therefore, with period $|\beta|\,|\alpha|$. Since $\alpha^{|\beta|}$ is a factor of ε and $|\beta|\,|\alpha|$ is a period of ε, the word ε is a factor of the word $\alpha^{|\beta|p}$ for a large enough natural number p. Hence, ε is a periodic word with period $|\alpha|$. As a result, $\alpha\delta$ also is a periodic word with period $|\alpha|$. However, this is impossible by Lemma 4, since δ is not a right extension of a power of α. □

4 Main Results

In this section, we show how to decide for a given finite deterministic automaton $A = (G, q_0, Q)$ whether the language $L(A)$ contains an infinite antichain or not with respect to the factor containment relation. Our solution is based on the analysis of the structure of cycles in G.

A cycle in G is any directed path with at least one edge in which the first and the last nodes coincide. A cycle is *simple* if its nodes are pairwise distinct (except for the first node being equal to the last node). Given a simple cycle C, we denote by $|C|$ the *length* of C, i.e. the number of nodes in C. For a node v of C, we denote by $w(C, v)$ the word of length $|C|$ obtained by reading the labels of the edges of C starting from the node v.

We distinguish between two basic cases: the case where G contains two different simple cycles that have at least one node in common and the case where all simple cycles of G are pairwise node disjoint.

Proposition 8. *Let $A = (G, q_0, Q)$ be a reduced finite deterministic automaton. If G contains two different simple cycles which have a node in common, then the language $L(A)$ contains an infinite antichain.*

Proof. Let C_1 and C_2 be two different simple cycles with a common node v. Since the cycles are different we may assume without loss of generality that the node of C_1 following v is different from the node of C_2 following v. As a result, the edges of C_1 and C_2 leaving v are labeled with different letters of the alphabet.

We denote $\alpha = w(C_1, v)^{|C_2|}$ and $\beta = w(C_2, v)^{|C_1|}$. The words α and β have the same length, but they differ in the first letter according to the above assumption.

Since A is reduced, there exist a directed path ρ from the start node to v and a directed path π from v to a terminal node. Therefore, every word of the form $w(\rho)\beta\alpha^i\beta w(\pi)$ ($i = 1, 2, \ldots$) belongs to the language $L(A)$. Since the words α and β have the same length and differ in the first letter, we conclude that β, and hence $w(\rho)\beta$, is not a left extension of a power of α. Similarly, the word $\beta w(\pi)$ is not a right extension of a power of α. Therefore, by Lemma 6, the language $L(A)$ contains an infinite antichain. □

From now on, we consider automata in which every two simple cycles are node disjoint. In this case, we decompose the set of nodes into finitely many subsets of simple structure, called metapaths.

A *metapath* consists of a number of node disjoint simple cycles, say $C_1, \ldots, C_t$ (possibly $t = 0$), and a number of directed paths $\rho_0, \ldots, \rho_t$ such that ρ_0 connects the start node q_0 to C_1, ρ_1 connects C_1 to C_2, ρ_2 connects C_2 to C_3, and so on, and finally, ρ_t connects C_t to a terminal node of the automaton. Let us observe that ρ_0 and ρ_t can be of length 0, while all the other paths are necessarily of length at least one, since the cycles are node disjoint.

We denote by $s(\rho_i)$ and $f(\rho_i)$ the first and the last node of ρ_i, respectively, and observe that for $i > 0$, $s(\rho_i)$ belongs to C_i, and for $i < t$, $f(\rho_i)$ belongs to C_{i+1}.

If $t = 0$, then the metapath contains no cycles and consists of the path ρ_0 alone. This path connects the start node q_0 to a terminal node of the automaton and no node of this path belongs to a simple cycle.

If $t > 0$, then $f(\rho_0), s(\rho_1), f(\rho_1), \ldots, s(\rho_{t-1}), f(\rho_{t-1}), s(\rho_t)$ are the only nodes of the paths $\rho_0, \rho_1, \ldots, \rho_t$ that belong to simple cycles.

For $i = 1, \ldots, t$, we denote by

– π_i the directed path from $f(\rho_{i-1})$ to $s(\rho_i)$ taken along the cycle C_i,
– γ_i the word $w(\pi_i)w(\rho_i)$,
– α_i the word $w(C_i, f(\rho_{i-1}))$.

Also, by γ_0 we denote the word $w(\rho_0)$.

Let τ be a metapath with t cycles, as defined above. The set of words accepted by this metapath can be described as follows: if $t = 0$ then $L(\tau) = \{\gamma_0\}$, and if $t > 0$ then

$$L(\tau) = \{\gamma_0 \alpha_1^{j_1} \gamma_1 \ldots \gamma_{t-1} \alpha_t^{j_t} \gamma_t : j_1, \ldots, j_t = 0, 1, \ldots\}.$$

Clearly, the set $T(A)$ of all metapaths is finite and

$$L(A) = \bigcup_{\tau \in T(A)} L(\tau).$$

It is also clear that $L(A)$ contains an infinite antichain if and only if $L(\tau)$ contains an infinite antichain for at least one metapath $\tau \in T(A)$.

If $t = 0$, the set $L(\tau)$ is finite and hence cannot contain an infinite antichain. In order to determine if $L(\tau)$ contains an infinite antichain for $t > 0$, we distinguish between the following three cases: $t = 1$, $t = 2$ and $t \geq 3$. In our analysis below we use the following simple observation:

Observation 1. *For $i = 1, \ldots, t-1$, the word γ_i is not a right extension of a power of α_i.*

The validity of this observation is due to the fact that the edge of ρ_i and the edge of C_i leaving vertex $s(\rho_i)$ must have different labels. On the other hand, we note that γ_0 may be a left extension of a power of α_1, while γ_t may be a right extension of a power of α_t.

Proposition 9. *Let τ be a metapath with exactly one cycle. Then $L(\tau)$ contains an infinite antichain if and only if*

– *neither γ_0 is a left extension of a power of α_1*
– *nor γ_1 is a right extension of a power of α_1.*

Proof. If γ_0 is a left extension of a power of α_1 or γ_1 is a right extension of a power of α_1, then $L(\tau)$ does not contain an infinite antichain by Lemma 5.

If neither γ_0 is a left extension of a power of α_1 nor γ_1 is a right extension of a power of α_1, then $L(\tau)$ contains an infinite antichain by Lemma 6. □

Proposition 10. *Let τ be a metapath with exactly two cycles. Then $L(\tau)$ contains an infinite antichain if and only if*

– *either γ_0 is not a left extension of a power of α_1*
– *or γ_2 is not a right extension of a power of α_2.*

Proof. Assume γ_0 is not a left extension of a power of α_1. From Observation 1 we know that γ_1 is not a right extension of a power of α_1. This implies that $\gamma_1\gamma_2$ is not a right extension of a power of α_1. Therefore, by Lemma 6, the set $\{\gamma_0\alpha_1^i\gamma_1\gamma_2 : i = 1, 2, \ldots\}$ contains an infinite antichain. Since this set is a subset of $L(\tau)$, we conclude that $L(\tau)$ contains an infinite antichain.

Suppose now that γ_2 is not a right extension of a power of α_2. By Observation 1, γ_1 is not a right extension of a power of α_1. Therefore, by Lemma 7, $\gamma_0\alpha_1^{|\alpha_2|}\gamma_1$ is not a left extension of a power of α_2. This implies, by Lemma 6, that the set $\{\gamma_0\alpha_1^{|\alpha_2|}\gamma_1\alpha_2^i\gamma_2 : i = 1, 2, \ldots\}$ contains an infinite antichain. Since this set is a subset of $L(\tau)$, we conclude that $L(\tau)$ contains an infinite antichain.

Finally, assume that γ_0 is a left extension of a power of α_1 and γ_2 is a right extension of a power of α_2. Then every word in $L(\tau)$ has the form $\sigma\alpha_1^i\gamma_1\alpha_2^j\delta$, where σ is a suffix of α_1, δ is a prefix of α_2, and i, j are natural numbers. Let $\varepsilon_1 = \sigma\alpha_1^{i_1}\gamma_1\alpha_2^{j_1}\delta$ and $\varepsilon_2 = \sigma\alpha_1^{i_2}\gamma_1\alpha_2^{j_2}\delta$ be two words of this form. Obviously, if these words are incomparable, then either $i_1 < i_2$ and $j_1 > j_2$ or $i_1 > i_2$ and $j_1 < j_2$. This implies that any antichain in $L(\tau)$ containing ε_1 can contain at most $i_1 + j_1$ words in $L(\tau)$ different from ε_1. Therefore the set $L(\tau)$ does not contain an infinite antichain. □

Proposition 11. *Let τ be a metapath with $t \geq 3$ cycles. Then $L(\tau)$ contains an infinite antichain.*

Proof. By Observation 1, γ_1 is not a right extension of a power of α_1, and hence, by Lemma 7, $\gamma_0\alpha_1^{|\alpha_2|}\gamma_1$ is not a left extension of a power of α_2. Also, by Observation 1, γ_2 is not a right extension of a power of α_2, and hence $\gamma_2 \ldots \gamma_t$ is not a right extension of a power of α_2. Therefore, by Lemma 6, the set $\{\gamma_0\alpha_1^{|\alpha_2|}\gamma_1\alpha_2^i\gamma_2 \ldots \gamma_t : i = 0, 1, 2, \ldots\}$ contains an infinite antichain. Since this set is a subset of $L(\tau)$, we conclude that $L(\tau)$ contains an infinite antichain. □

We summarize the above discussion in the following final statement which is the main result of the paper.

Theorem 12. *Given a deterministic finite automaton A one can decide in polynomial time whether the language accepted by A contains an infinite antichain with respect to the factor containment relation or not.*

Proof. Since the automaton A is finite, the question of the existence of infinite antichains in $L(A)$ is decidable by Propositions 8, 9, 10, 11. Now we sketch the proof of polynomial-time solvability.

First, we identify strongly connected components in G, which can be done in polynomial time. If at least one strongly connected component contains a simple cycle and is different from the cycle (i.e. contains at least one edge outside of the cycle), then it necessarily contains two cycles with a common node, in which case $L(A)$ contains an infinite antichain by Proposition 8.

If each of the strongly connected components of G is a simple cycle (or a single vertex without loops), then the cycles of G are pairwise node disjoint. In this case, we construct an auxiliary acyclic graph G' by contracting each simple cycle into a single node, called *cyclic node*. Each metapath in G corresponds to a directed path in G'.

First, we check if G' contains a directed path containing at least 3 cyclic nodes. This can be done for G' in cubic time. If such a path exists, then by Proposition 11 $L(A)$ necessarily contains an infinite antichain.

In order to check if G contains a metapath with precisely two cycles and satisfying conditions of Proposition 10, we choose in G' an ordered pair of cyclic nodes c_1, c_2 and delete from the graph all other cyclic nodes, obtaining in this way the graph G''. The nodes c_1 and c_2 correspond to the cycles C_1 and C_2 in G. We assume that in G'' there is a directed path ρ_0 connecting q_0 to c_1, a directed path ρ_1 connecting c_1 to c_2 and a directed path ρ_2 connecting c_2 to a terminal node. Any three such paths must be node disjoint (except, of course, c_1 and c_2), since otherwise a directed cycle arises. Therefore, together with C_1 and C_2 any three such paths form a metapath in G. Our task is to verify if there is a triple (ρ_0, ρ_1, ρ_2) that defines a metapath satisfying conditions of Proposition 10.

Let us observe that every choice of ρ_0 uniquely defines the word α_1 (inscribed in C_1) which we denote by $\alpha_1^{\rho_0}$. Therefore, in order to verify the first of the two conditions of Proposition 10 we need to solve the following problem.

(1) Determine if G'' contains a path ρ_0 connecting q_0 to c_1 such that the corresponding word $\gamma_0 = w(\rho_0)$ is not a left extension of a power of $\alpha_1^{\rho_0}$.

The second of the two conditions of Proposition 10 involves the word α_2, which is defined by the path ρ_1. However, it turns out that this condition can be verified without specifying this path. In order to show this, let us associate with every path ρ_2 connecting c_2 to a terminal node the word $\alpha_2^{\rho_2}$ defined as $w(C_2, s(\rho_2))$, i.e. $\alpha_2^{\rho_2}$ is the word of length $|C_2|$ inscribed in C_2 starting at the first node of ρ_2. Then $\gamma_2 = w(\pi_2)w(\rho_2)$ is not a right extension of a power of α_2 if and only if $w(\rho_2)$ is not a right extension of a power of $\alpha_2^{\rho_2}$. Therefore, in order to verify the second of the two conditions of Proposition 10 we need to solve the following problem.

(2) Determine if G'' contains a path ρ_2 connecting c_2 to a terminal node such that $w(\rho_2)$ is not a right extension of a power of $\alpha_2^{\rho_2}$.

To solve problem (1), we denote by n the number of nodes of G'' and by N_i the set of nodes of G'' for which there exists a directed path of length i from q_0

($N_0 = \{q_0\}$). We observe that the sets N_i are not necessarily disjoint, each of them contains at most n nodes and for $i > n$, the sets N_i are empty (since G'' is acyclic). Consider a vertex $x \in N_i$ with $i > 0$ and assume there is a directed path connecting x to $c_1 \in N_j$ with $j > i$ (which can be easily verified). We check if at least two edges coming to x from the vertices of N_{i-1} are labelled with different letters of the alphabet. If this is the case, then clearly there is a path ρ_0 connecting q_0 to c_1 through x such that the corresponding word $\gamma_0 = w(\rho_0)$ is not a left extension of a power of $\alpha_1^{\rho_0}$. If for each i and for each vertex $x \in N_i$ all edges coming to x from N_{i-1} have the same label and this label coincides with the respective letter of $\alpha_1^{\rho_0}$ (counted cyclically from the end of $\alpha_1^{\rho_0}$), then problem (1) has no positive solution. It is not difficult to see that the overall time complexity of problem (1) is polynomial in the number vertices of G.

To solve problem (2), for each vertex v of C_2 we check if v can be connected to a terminal node by at least two different paths. If this is the case, then at least one of them is a path ρ_2 such that the word $w(\rho_2)$ is not a right extension of a power of $\alpha_2^{\rho_2}$, because the edges leaving the node where the two paths split must be labeled differently. If for each $v \in C_2$ there is at most one path ρ_2 connecting v to a terminal node and the word $w(\rho_2)$ is a right extension of a power of $\alpha_2^{\rho_2}$, then problem (2) has no positive solution. Clearly, problem (2) can be solved in polynomial time too.

If at least one of the two problems, say (1), has a positive solution, then, in addition to this solution, we find an arbitrary directed path connecting c_1 to c_2 and an arbitrary directed path connecting c_2 to a terminal node. Together with the cycles C_1 and C_2 these three paths form a metapath τ in G such that $L(\tau)$ contains an infinite antichain by Proposition 10.

The above arguments show that in polynomial time we can check if G has a metapath τ with precisely two cycles such that $L(\tau)$ contains an infinite antichain. If this is not the case, we need to check if G contains a metapath with exactly one cycle satisfying conditions of Proposition 9. This can be done by solving for each cyclic node of G' two problems similar to problems (1) and (2). $\square$

References

1. Atkinson, M.D., Murphy, M.M., Ruškuc, M.: Partially well-ordered closed sets of permutations. Order 19(2), 101–113 (2002)
2. Brignall, R., Ruškuc, N., Vatter, V.: Simple permutations: decidability and unavoidable substructures. Theoretical Computer Science 391(1-2), 150–163 (2008)
3. Cherlin, G.L., Latka, B.J.: Minimal antichains in well-founded quasi-orders with an application to tournaments. J. Comb. Theory, Ser. B 80(2), 258–276 (2000)
4. Crochemore, M., Mignosi, F., Restivo, A.: Automata and forbidden words. Inf. Process. Lett. 67(3), 111–117 (1998)
5. Ding, G.: Subgraphs and well-quasi-ordering. J. Graph Theory 16(5), 489–502 (1992)
6. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)

7. Hine, N., Oxley, J.: When excluding one matroid prevents infinite antichains. Advances in Applied Mathematics 45(1), 74–76 (2010)
8. Korpelainen, N., Lozin, V.V.: Two forbidden induced subgraphs and well-quasi-ordering. Discrete Mathematics 311(6), 1813–1822 (2011)
9. Kruskal, J.B.: The theory of well-quasi-ordering: a frequently discovered concept. J. Comb. Theory, Ser. A 13(3), 297–305 (1972)
10. de Luca, A., Varricchio, S.: Well quasi-orders and regular languages. Acta Inf. 31(6), 539–557 (1994)
11. Robertson, N., Seymour, P.: Graph Minors. XX. Wagner's conjecture. J. Comb. Theory, Ser. B 92(2), 325–357 (2004)
12. Spielman, D.A., Bóna, M.: An infinite antichain of permutations. The Electr. J. Comb. 7 (2000)

On the Construction of a Family of Automata That Are Generically Non-minimal

Parisa Babaali[1] and Christopher Knaplund[2]

[1] Department of Mathematics and Computer Science
York College of the City University of New York
94-20 Guy R. Brewer Blvd, Jamaica, NY, USA
pbabaali@york.cuny.edu

[2] Department of Mathematics
Graduate Center of the City University of New York
365 5th Avenue, New York, NY, USA
cknaplund@math.gc.cuny.edu

Abstract. One way to generate an accessible deterministic finite automaton is to first generate a spanning tree and then complete it to an automaton. We introduce the ideas of a sequential automaton, that are automata with sequential trees as breadth-first spanning subtrees. We introduce the concept of elementary equivalent states and explore combinatorial properties of non-minimal sequential automata. We then show that minimality is negligible among sequential automata by calculating the probability that an automaton has two elementary equivalent states and showing that this probability approach 1 as the size of the automaton increases.

Keywords: minimal automata, asymptotic density, random generation, sequential tree.

1 Introduction

Deterministic Finite Automata (DFA) are the simplest form of computation, composed of a finite number of states Q, transition between states $\delta : Q \times \Sigma \rightarrow Q$, the start state,$i$ and the set of final states F. An accessible DFA (ADFA) is one that the start state can reach every other state. There are a number of ways to generate an ADFA with n states. In 1973, Harary and Palmer [12] enumerated isomorphic automata with output functions as certain ordered pairs of functions. Harrison [13] considered the enumeration of non-isomorphic DFAs and connected DFAs up to a permutation of the alphabet symbols. Korshunov [15] has enumerated the number of non-isomorphic strongly connected finite automata and with the same criteria Robinson [20] has counted the number of strongly connected automata. Domaratzki et al. [10] have proposed a lower bound for the number of accessible deterministic finite automata (ADFAs) over an alphabet of size k. Nicaud [18] in 2000, and Champarnaud and Paranthoen [9] in 2005 presented a method for randomly generating ADFAs using spanning trees. Nijenhuis and Wilf [19] introduced a recursive method to generate DFAs at random, which was later systematized by Flajolet, Zimmermann and Van Custem [11]. This method requires a significant memory space however.

Bassino and Nicaud [5,7] showed that the number of ADFAs is $\Theta(n2^nS(kn,n))$, where $S(kn,n)$ is the Stirling number of the second kind. Using breadth-first spanning trees, Almeida et. al [1] have also proposed efficient algorithms to generate ADFAs at random and confirmed the previous result.

There is also an interest in generating a minimal automaton with n states which is an open question. One way to generate a minimal automaton is to generate an accessible DFA at random and use a rejection algorithm, to decide if it is minimal, assuming that the asymptotic density of the minimal automata exists and is constant. Recently Bassino, et al [6] have shown that the asymptotic density of minimal automata in the set all ADFAs of size n is constant. They have shown that for an alphabet of size k, the proportion of minimal automata is $e^{-\frac{1}{2}c_k n^{-k+2}}$ where $c_k = \frac{1}{2}\omega_k^k$, and ω_k is a solution of $-k\omega_k = \ln(1-k)$. For $k=2$ this density is about 0.8532.

We have used the breadth first spanning tree to generate families of automata. In such families, the number of automata having a given tree as a breadth-first subtree varies widely, for instance there are trees, *degenerate trees* for which there are $2^n nn!$ automata in the span of the tree, and there are trees, *full binary trees* with $2^n n^{n+1}$ automata in the span of the tree. In addition to that, these families behave very differently in terms of minimality, i.e. there are trees for which minimality is generic in the span of the tree and there are those for which minimality is negligible, however for most trees the proportion of minimal automata hovers around the bound of Bassino et al [6].

In our investigation of automata generation over an alphabet of size 2, we have constructed a family of automata that are almost always minimal [4]. While studying those that are minimal is our objective, we have come across a family of automata that are always non-minimal. In this paper we construct a family of trees and hence automata in the span of each tree for which the probability that the automaton is minimal approaches 0 as the number of states approach infinity.

We use a breadth-first spanning subtree to construct the automata we call sequential automata. This paper is organized as follows: section 2 includes some definitions and notations related to string representation of automata and trees, and minimality. In section 3 we introduce the notion of a sequential automaton and study combinatorial properties of such automata when non-minimal, we introduce elementary equivalence of states (similar to M-motif in [6]), and find the asymptotic density of a sequential automaton having two or more elementary equivalent states. We calculate the conditional probability of being non-minimal using a recursive equation, and we present some experimental results obtained from Monte-Carlo simulations on such automata and show that they agree with our results.

2 Definitions and Notation

2.1 Asymptotic Density

A size function on a set $\mathcal{A}$ is a map $s : \mathcal{A} \to \mathbb{N}$ such that for all $n \in \mathbb{N}$; $s^{-1}(n)$ is a finite set. A stratification of $\mathcal{A}$ into balls is an increasing sequence of subsets of $\mathcal{A}$ with $\mathcal{A}_0 \subseteq \mathcal{A}_1 \subseteq \mathcal{A}_2 \cdots$ where each $\mathcal{A}_i$ is finite and $\bigcup_{i=1}^{\infty} \mathcal{A}_i = \mathcal{A}$. Hence given a size function s on $\mathcal{A}$, one can find a stratification of $\mathcal{A}$ by $\mathcal{A}_0 = s^{-1}(\{0\})$ and $\mathcal{A}_i = s^{-1}(\{0, 1, \cdots, i\})$. We can also form spheres of radius n defined as $S_n = \mathcal{A}_n - \mathcal{A}_{n-1}$.

Let $\mathcal{M}$ be a subset of $\mathcal{A}$. Define the upper (respectively lower) spherical asymptotic density of $\mathcal{M}$ to be

$$\bar{\rho} = \limsup_{n\to\infty} \frac{|\mathcal{M} \cap S_n|}{|S_n|}, \text{ and } \underline{\rho} = \liminf_{n\to\infty} \frac{|\mathcal{M} \cap S_n|}{|S_n|}$$

We say that $\mathcal{M}$ has a spherical asymptotic density ρ if the limit exist and $\bar{\rho} = \underline{\rho}$. We say $\mathcal{M}$ is generic if $\rho = 1$ and is negligible if $\rho = 0$. Note that $\rho_n = \dfrac{|\mathcal{M} \cap S_n|}{|S_n|}$ can also be viewed as $\mathbb{P}(\mathcal{M})$ in S_n with respect to the uniform measure on S_n.

2.2 Automata, Trees, String Representation and Random Generation

A deterministic finite automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of *states*, Σ is a finite *input alphabet*, $q_0 \in Q$ is the *initial state*, $F \subset Q$ is the set of *final states*, and δ is the *transition function* mapping $Q \times \Sigma$ to Q. Extend δ by defining $\delta(q, aw) = \delta(\delta(q, a), w)$. An ADFA (accessible deterministic finite automaton) is an initially connected DFA in which there is a directed path from the start state, q_0 to every other state. A transition structure(TS) is an automaton (Q, Σ, δ, q_0) with no final states.

In this section we also present a canonical representation of a transition structure and its respective spanning tree which is very useful in enumeration, generation and analysis of automata. This presentation has been discussed and used by Almeida et al. in [1] and Babaali in [2]. Any ADFA, A with n states, can be decomposed into its breadth-first spanning tree with n nodes and the remaining transitions. This decomposition will lead to an ordering of states of A, and hence a numbering of states, from 1 to n. This numbering is unique when we fix an order on Σ. In our paper we fix the ordering $a < b$ in $\Sigma = \{a, b\}$. We also let the start state to always be state number 1.
Considering the spanning tree T, it can be represented by a binary sequence β_T of length $2n + 1$, where $\beta_T(0) = 1$; $\beta_T(i) \in \{0, 1\}$. For $1 \le k \le n$, define the *binary representation* β_T of a tree T by

$$\beta_T(2k-1) = \begin{cases} 1 & \text{if there is an edge leaving state } k \text{ with label } a, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$\beta_T(2k) = \begin{cases} 1 & \text{if there is an edge leaving state } k \text{ with label } b, \\ 0 & \text{otherwise.} \end{cases}$$

Notice that by looking at $\beta_T(i)$, we can determine if there is an edge leaving state $\left\lceil \frac{i}{2} \right\rceil$, labeled by alphabet letters a or b. Similar representations have been used to generate random binary trees, as shown in the survey by Mäkinen in [16]. One can show that each β_T has precisely n 1s, and $n+1$ 0s. This representation also has the property that each initial segment w of β_T has the property $|w|_0 < 1 + |w|_1$. An example of such a representation is shown in figure 1.

Example 1. Consider the automaton A_T and its breadth-first spanning tree T shown below. Using the definition of the binary representation of a tree, $\beta_T =$ 101110000.

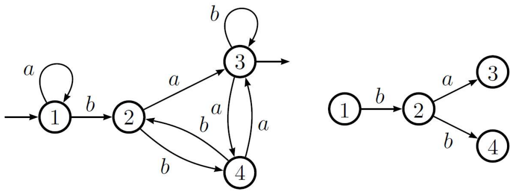


Fig. 1. An automaton and its spanning tree T

For a given tree T, the *span of* T, $\mathcal{S}(T)$ is the set of all automata having T as a breadth-first spanning subtree. Now let us count the number of automata in $\mathcal{S}(T)$ for a given T. Define a new sequence K_T to be the *difference sequence* of T, which is defined as the number of zeroes between any two consecutive 1's in β_T. Hence $K_T = (k_1, k_2, \cdots, k_n)$ is a sequence of non-negative integers of length n, with the following properties:

1) $k_n \leq n+1$, 2) $0 \leq \sum_{j=1}^{i} k_j \leq i$ for $1 \leq i \leq n-1$, and 3) $\sum_{i=1}^{n} k_i = n+1$. The string representation and the difference sequence of a tree are very helpful tools in counting $|\mathcal{S}(T)|$, the number of automata having this tree as a breadth-first spanning subtree. That is:

$$|\mathcal{S}(T)| = 2^n \prod_{i=1}^{n} i^{k_i} \tag{1}$$

The factor of 2^n is counting the different combinations of subsets of final states, and $\prod_{i=1}^{n} i^{k_i}$ is the number of transition structures having T as a spanning subtree. For example for the tree of figure 1, the difference sequence is $K_T = (1, 0, 0, 4)$ and there are $|\mathcal{S}(T)| = 2^4 \times 1^1 \times 2^0 \times 3^0 \times 4^4$ automata having T as a spanning subtree. More details on this method and the associated proof can be found at [3].

2.3 Minimal Automata

Let A be a DFA, the language accepted by a A is defined as $L(A) = \{w \in \Sigma^* | \delta(q_0, w) \in F\}$. Two DFAs are equivalent if they accept the same language. An automaton is *minimal* if it is the automaton with the smallest number of states accepting a given regular language. Minimal DFAs are unique up to a renaming of the states, and the minimal automaton of a regular language can be found using a number of algorithms. There are a number of algorithms that compute the minimal automaton of A by computing the coarsest partition that saturates F. Hopcroft [14], Moore [17] and Brzozowski [8] are among the most studied algorithms.

A partition $\mathcal{P} = \{P_1, P_2, \cdots, P_m\}$of Q is a set of disjoint subsets of Q with $Q = \bigcup_{i=1}^m P_i$. We say that $\mathcal{P}$ saturates F if F is a union of classes of $\mathcal{P}$. Two states $p, q \in P_i$ in a given automata A are indistinguishable or equivalent states if for all $w \in \Sigma^*$: $\delta(p, w) \in F \iff \delta(q, w) \in F$

Clearly p is distinguishable from q if there is a string w such that $\delta(p, w)$ is in F but $\delta(q, w)$ is not. In a minimal automaton all states are distinguishable. Let $p \sim q$ if p and q are indistinguishable states. Then $\sim$ is an equivalence relation on Q which leads to a partition of equivalent states in Q. In more details, an automaton can be minimized with identification of equivalent states in the coarsest partition of Q that saturates F.

Definition 2. *Let $A = (Q, \Sigma, \delta, q_0, F)$ be an ADFA, with $i, j \in Q$ state of A which are both final or non-final. Then we say that i and j are elementary equivalent and we denote it by $i \sim_e j$ if any of the following condition holds*

1. $\delta(i, \sigma) = \delta(j, \sigma)$ *for all* $\sigma \in \Sigma$;
2. $\delta(i, \sigma_1) = j$, $\delta(j, \sigma_1) = i$ *and* $\delta(i, \sigma) = \delta(j, \sigma)$ *for all* $\sigma \in \Sigma - \{\sigma_1\}$;
3. $\delta(i, \sigma_1) = i$, $\delta(j, \sigma_1) = j$ *and* $\delta(i, \sigma) = \delta(j, \sigma)$ *for all* $\sigma \in \Sigma - \{\sigma_1\}$.

A visual description of elementary equivalent states on an alphabet of size 2 is shown in figure 2. Note that the first condition is an M-motif presented in [6].

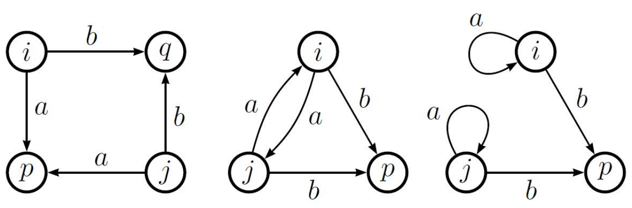


Fig. 2. Elementary equivalent states on $\Sigma = \{a, b\}$

Lemma 3. *If A is a DFA with some states i and j with $i \sim_e j$ then i and j are indistinguishable states and hence A is not minimal.*

3 Sequential Automata

In this section we study classes of automata $\mathcal{S}(T)$ for which the conditional asymptotic density of an automaton being minimal approaches 0 as the number of states of the automaton increases. That is to say that for very large values of $|Q| = n$, the automaton is very likely to be non-minimal.

Definition 4. *A binary tree T is* sequential *if it has a string representation of the form $\beta_T = 111a_1a_2 \cdots a_{k-1}0000$ where $a_i = 1100$ or 0011, i.e. at every level, there is exactly one node that has no children and one that has two children. An automaton is sequential, if it has a sequential tree T as a breadth first spanning subtree. That is to say $A \in \mathcal{S}(T)$.*

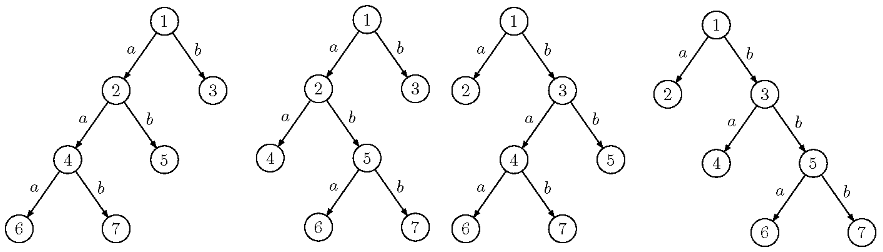


Fig. 3. All Sequential trees of size 7

Examples of sequential trees of size 7 are shown in figure 3. Using the recurrence equation $c_n = 2c_{n-2}$ with $c_3 = 1$, there are $c_n = 2^{\frac{n-3}{2}}$ sequential trees with n nodes. Recall that such tree has an odd number of states, say $n = 2k + 1$. A sequential automaton and its breadth first spanning tree are shown in figure 4:

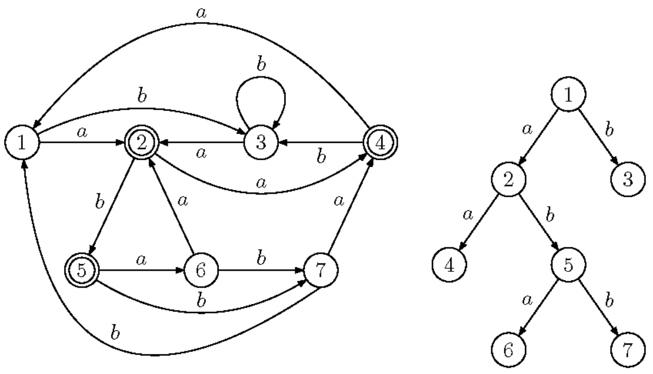


Fig. 4. A Sequential automaton and its breadth first spanning tree

3.1 Probability Measures and the Number of Sequential Automata

Let us define the uniform probability measure on $\mathcal{S}(T)$ for a given sequential tree. In order to do this we need to count the number of automata in $\mathcal{S}(T)$. Studying the possible difference sequences for sequential trees of size $n = 2k+1$, it is easy to see that the sequential tree with the smallest number of automata is $T_b = 111\underbrace{0011\cdots0011}_{k-1\text{ times}}0000$, this is the tree where the right child always has 2 children and the left child has no children. The number of automata in $\mathcal{S}(T_b)$ is:

$$|\mathcal{S}(T_b)| = 2^n \times 3^2 \times 5^2 \times 7^2 \cdots \times (2k+1)^2 \times (2k+1)^2 = (n!!)^2\, n^2$$

The sequential tree with the largest number of automata is,

$$T_a = 111\underbrace{1100\cdots1100}_{k-1 times}0000$$

the tree where the right child always has no children and the left child has two children. The number of automata in $\mathcal{S}(T_a)$ can be calculated as:

$$|\mathcal{S}(T_a)| = 2^n \times 5^2 \times 7^2 \times 9^2 \cdots \times (2k+1)^2 \times (2k+1)^4 = (n!!)^2\, \frac{n^4}{9}$$

Hence we have a bound on the number of automata in $|S(T)|$

$$2^n\,(n!!)^2\, n^2 \leq |\mathcal{S}(T)| \leq 2^n\,(n!!)^2\,\frac{n^4}{9}$$

For a subset $\mathcal{E} \subset \mathcal{S}(T)$ define

$$\mathbb{P}(\mathcal{E}) = \mathbb{P}(A \in \mathcal{E}) = \frac{|\mathcal{E}|}{|\mathcal{S}(T)|}$$

One way an automaton to be non-minimal, is to have states that are elementary equivalent. An interesting and powerful property of a random sequential automaton is the number of configurations for a state to be elementary equivalent to another state. In the next section we will compute this number in more detail and show that there are many configurations for which the automaton has two or more elementary equivalent states. We only need to find these probabilities for $\mathcal{S}(T_a)$ and $\mathcal{S}(T_b)$, since for all other sequential automata these probabilities are bounded by those of $\mathcal{S}(T_a)$ and $\mathcal{S}(T_b)$.

3.2 Elementary Equivalent States for Automata in $\mathcal{S}(T_a)$

There are two types of state in a sequential tree, 1)those that are odd numbered, or leaves in the spanning tree with a degree of freedom of 2, and 2) those that are even numbered, which have a degree of freedom of 0. Recall that for a state to be elementary equivalent to another, at least one of the states have to be a leaf in the spanning tree. Knowing this we will find the probability that an odd-numbered state is elementary equivalent to another state.

Case 1. Let $\mathcal{E}_{i,l}$ be the set of automata in $\mathcal{S}(T_a)$ in which state $2i+1 \sim_e 2l$ for some $2l \leq 2i$. We will estimate $\mathbb{P}(\mathcal{E}_{i,l})$. In this case, since $\delta(2l, a) = 2l+2$ and $\delta(2l, b) = 2l+3$, the choices for $\delta(2i+1, \sigma)$ are limited to that of $\delta(2l, \sigma)$ for $\sigma = a, b$. The next important point is that states $2i+1$ and $2l$ have the same parity, that is they are both final or non-final. Since all other remaining states q have freedom in the number of choices for $\delta(q, \sigma)$, we have

$$\mathbb{P}(\mathcal{E}_{i,l}) = \frac{2^{n-1} \times 5^2 \times 7^2 \times \cdots \times 1^2 \times \cdots \times (2k+1)^6}{2^n \times 5^2 \times 7^2 \cdots \times (2i+3)^2 \times \cdots \times (2k+1)^6} = \frac{1}{2}\,\frac{1}{(2i+3)^2}$$

Case 2. Let $\mathcal{E}_{i,j}$ be the set of automata in $\mathcal{S}(T_a)$ in which state $2i+1 \sim_e 2j+1$ for some $j < i$. In this case, $df(2i+1) = df(2i+1) = 2$, which implies that $\delta(2j+1, \sigma)$ has $2j+3$ choices, particularly $\delta(2j+1, \sigma)$ may be $2j+1$. In this case $\delta(2i+1, \sigma)$ can be both $2i+1$ or $2j+1$. This is the second case in Lemma 2. Hence

$$\mathbb{P}(\mathcal{E}_{i,j}) = \prod_{\sigma} \mathbb{P}\Big(2i+1 \sim_e 2j+1 \;\Big|\; \delta(2j+1, \sigma) = 2j+1\Big) +$$
$$\mathbb{P}\Big(2i+1 \sim_e 2j+1 \;\Big|\; \delta(2j+1, \sigma) \neq 2j+1\Big)$$
$$= \left(\frac{1}{2}\,\frac{2}{(2i+3)}\frac{1}{(2j+3)} + \frac{1}{2}\,\frac{1}{(2i+3)}\frac{2j}{(2j+3)}\right)^2 = \left(\frac{j+1}{(2j+1)(2i+1)}\right)^2$$

Finally we are interested in $\mathbb{P}(\mathcal{E}_i) = \mathbb{P}\left(\bigcup_k \mathcal{E}_{i,k}\right)$. Recall

$$\mathbb{P}(\mathcal{E}_i) = \mathbb{P}\left(\bigcup_{k=1}^{i-1} \mathcal{E}_{i,k}\right) =$$
$$\sum_{k=1}^{i-1} \mathbb{P}\left(\mathcal{E}_{i,k}\right) - \underbrace{\sum_{k,j} \mathbb{P}\left(\mathcal{E}_{i,k} \cap \mathcal{E}_{i,j}\right)}_{(*)} + \sum_{n,m,j} \mathbb{P}\left(\mathcal{E}_{i,n} \cap \mathcal{E}_{i,m} \cap \mathcal{E}_{i,j}\right) - \cdots$$

First note that $\mathcal{E}_{i,j}$ are independent and hence $\mathbb{P}\left(\mathcal{E}_{i,n} \cap \mathcal{E}_{i,m}\right) = \mathbb{P}\left(\mathcal{E}_{i,n}\right)\mathbb{P}\left(\mathcal{E}_{i,m}\right)$. We can see that the contribution of additional term of intersection of these events are extremely improbable, hence negligible. To see this, first note that when 3 or more states are elementary equivalent, at most one of them can be an even numbered state. Let's study the smallest case (*), and suppose that $2i+1 \sim_e 2k \sim_e 2j+1$.

$$\sum_{k \leq j \leq i} \mathbb{P}\left(\mathcal{E}_{i,k} \cap \mathcal{E}_{i,j}\right) = \sum_{k \leq j \leq i} \mathbb{P}\left(\mathcal{E}_{i,k}\right)\mathbb{P}\left(\mathcal{E}_{i,j}\right) = \sum_{j=1}^{i-1}\sum_{k=1}^{j-1} \frac{1}{2(2i+3)^2}\,\frac{1}{2(2j+3)^2}$$

$$= \frac{1}{2(2i+3)^2} \sum_{j=1}^{i-1} \frac{j-1}{2(2j+3)^2} \leq \frac{1}{2^4(2i+3)^2} \sum_{j=1}^{i-1} \frac{1}{j+3/2}$$

$$\leq \frac{1}{2^4(2i+3)^2} \sum_{j=1}^{i-1} \frac{1}{j} \sim \frac{\ln(i-1)+C}{2^4(2i+3)^2} \text{ for some constant C}$$

A very similar argument shows that when $2i+1 \sim_e 2k+1 \sim_e 2j+1$, the term obtained from the intersection has order $\frac{1}{2^4(2i+3)^2} \sum_{j=1}^{i-1} \frac{\log(2j+1)}{(2j+3)^2}$. The sum of both these terms is very small. With the intersection of m terms a factor of $\frac{1}{2^{m+2}(2i+3)^2}$ is added to each term of the sum. A similar argument to the above can be used to show that this term's contribution to the sum of $P(E_{i,j})$ is of negligible order. Hence the magnitude of what is taken away from $\sum_k \mathbb{P}(\mathcal{E}_{i,k})$ is very small. Now we have:

$$\mathbb{P}(\mathcal{E}_i) = \underbrace{\frac{1}{2(2i+3)^2}}_{P(2i+1 \sim_e 1)} + \underbrace{\sum_{l=1}^{i-1} \frac{1}{2(2i+3)^2}}_{P(2i+1 \sim_e l)} + \sum_{j=1}^{i-1} \left(\frac{j+1}{(2j+3)(2i+3)} \right)^2 - \text{small term}$$

$$\sim \frac{i}{2(2i+3)^2} + \frac{1}{(2i+3)^2} \sum_{j=1}^{i-1} \left(\frac{j+1}{2j+3} \right)^2 \leq \frac{3i+\ln(i)+C}{4(2i+3)^2} \leq \frac{4i}{4(2i+3)^2}$$

for $i < n$. Note that there is an exception for when $i = n$. In this case

$$\mathbb{P}(\mathcal{E}_n) \sim \frac{n}{2(2n+1)^2} + \frac{1}{(2n+1)^2} \sum_{j=1}^{n-1} \left(\frac{j+1}{2j+3} \right)^2$$

3.3 Elementary Equivalent States for Automata in $\mathcal{S}(T_b)$

A similar argument can be used to find $\mathbb{P}(\mathcal{E}_i$ in $\mathcal{S}(T_b)$. Let $\mathcal{E}_{2i,l}$ be the set of automata in $\mathcal{S}(T)$ in which state $2i \sim_e q$ for some $q \leq 2i-1$.

Case 1. When $2i \sim_e 2l-1$ for some $l \leq i$. In this case:

$$\mathbb{P}(\mathcal{E}_{i,l}) = \frac{1}{2} \frac{1}{(2i+1)^2}$$

Case 2. When $2i \sim_e 2j$ for some $j < i$. In this case, since $\delta(2j,a)$ has $2j+1$ choices, particularly $\delta(2j,\sigma)$ may be $2j$, hence $\delta(2i,\sigma)$ are very limited:

$$\mathbb{P}(\mathcal{E}_{2i,2j}) = \prod_{\sigma} \mathbb{P}\Big(2i \sim_e 2j \Big| \delta(2j,\sigma) = 2j\Big) + \mathbb{P}\Big(2i \sim_e 2j \Big| \delta(2j,\sigma) \neq 2j\Big)$$

$$= \left(\frac{1}{2} \, \frac{2}{(2i+1)} \frac{1}{(2j+1)} + \frac{1}{2} \, \frac{1}{(2i+1)} \frac{2j}{(2j+1)} \right)^2 = \left(\frac{j+1}{(2j+1)(2i+1)} \right)^2$$

Again, we can assume that the contribution of the intersection of terms in $P(\mathcal{E}_i)$ is negligible. Hence

$$\mathbb{P}(\mathcal{E}_i) = \mathbb{P}\left(\bigcup_k \mathcal{E}_{i,k} \right) \sim \sum_{l=1}^{i-1} \frac{1}{2(2i+1)^2} + \frac{1}{2(2i+1)^2} + \sum_{j=1}^{i-1} \left(\frac{j+1}{(2j+1)(2i+1)} \right)^2$$

We can bound $\mathbb{P}(\mathcal{E}_i)$ by:

$$\frac{2i+1}{4(2i+1)^2} \leq \mathbb{P}(\mathcal{E}_i) \leq \frac{4i+1}{4(2i+1)^2} \tag{2}$$

3.4 Minimality in $\mathcal{S}(T)$ for a Sequential Tree

In this section we will find the probability that an automaton is non-minimal, only by considering the occurrence of states that are elementary equivalent. As it turns out, this event is probable enough that for large values of n, it is a generic set in $\mathcal{S}(T)$. Consider an automaton in $S(T)$, with $2n+1$ states, and let $P_n = \mathbb{P}(\bigcup_{i=1}^{n} \mathcal{E}_i)$, Recall that E_i is the event that state $2i+1 \sim_e q$ for some state $q \in Q$.

$$P_k = \mathbb{P}(\bigcup_{i=1}^{n} \mathcal{E}_i) = \mathbb{P}(\bigcup_{i=1}^{n-1} \mathcal{E}_i \cup \mathcal{E}_n) = P_{n-1} + \mathbb{P}(\mathcal{E}_n) - P_{n-1}\mathbb{P}(\mathcal{E}_n)$$

Let $a_n = \mathbb{P}(\mathcal{E}_n)$, in this case the recurrence equation is

$$P_n = P_{n-1}(1 - a_n) + a_n \tag{3}$$

We can solve P_k for a given a_k. For instance when $a_k = \dfrac{4k+1}{4(2k+1)^2}$, then equation 3 has a solution of the form:

$$P_k = 1 - \frac{|\Gamma(k+\omega)|^2}{|\Gamma(k+\frac{1}{2})|^2} \text{ where } \omega \text{ is a solution of equation } \omega^2 + \tfrac{3}{4}\omega + \tfrac{3}{16} = 0$$

Let $l = \dfrac{|\Gamma(k+\omega)|^2}{|\Gamma(k+\frac{1}{2})|^2}$, we show that $l \to 0$ as $n \to \infty$.

$$l = \frac{|\Gamma(k+\omega)|^2}{|\Gamma(k+\frac{1}{2})|^2} = \frac{|\omega+n-1|^2}{|\frac{1}{2}+n-1|^2} \, \frac{|\omega+n-2|^2}{|\frac{1}{2}+n-2|^2} \cdots \frac{|\omega+1|^2}{|\frac{1}{2}+1|^2} \, \frac{|\omega|^2}{|\frac{1}{2}|^2} \frac{|\Gamma(\omega)|^2}{|\Gamma(\frac{1}{2})|^2}$$

$$= \prod_{i=1}^{n} \frac{(n-i)^2 + \frac{3}{4}(n-i) + \frac{3}{16}}{(n-i)^2 + (n-i) + \frac{4}{16}} \, \frac{|\Gamma(\omega)|^2}{|\Gamma(\frac{1}{2})|^2}$$

$$\log(l) = \sum_{i=0}^{n} \log\left(\frac{(n-i)^2 + \frac{3}{4}(n-i) + \frac{3}{16}}{(n-i)^2 + (n-i) + \frac{4}{16}}\right) = \sum_{i=1}^{n} \log\left(1 - \frac{i + \frac{1}{4}}{4i^2 + 4i + 1}\right)$$

The behavior of this series $\log(l)$ is the same as $\log\left(1 - \frac{1}{i}\right)$, and hence the series diverge. This proves

$$\lim_{n\to\infty} P_n = 1 - \lim_{n\to\infty} \frac{|\Gamma(n+\omega)|^2}{|\Gamma(n+\frac{1}{2})|^2} = 1 - 0 = 1$$

It can be shown that for all other sequences $a_n = \mathbb{P}(\cup_{i=1}^{n} E_i)$, P_n has a solution of the form $P_n = 1 - \frac{|\Gamma(n+\omega)|^2}{|\Gamma(n+\frac{1}{2})|^2}$ that converges to 1, however the convergence is extremely slow. The following theorem is proved.

Theorem 5. *Let T be a sequential tree. Then the conditional asymptotic density of $\mathcal{M}$ in $\mathcal{S}(T)$ is zero. That is*

$$\lim_{n\to\infty} \mathbb{P}(\mathcal{M}|\mathcal{S}(T)) = 0$$

In other words minimality is a negligible property among degenerate automata.

4 Conclusion

In this paper we have shown that minimality is a negligible property among sequential automata, i.e. those automata with a sequential tree as a breadth first subtree, however the convergence is extremely slow. We have confirmed this result experimentally by generating a large number of sequential automata for each size shown in table 1. The automata were generated uniformly randomly, and the proportion of minimal automata was calculated using Hopcroft's algorithm. The numerical values are presented in table 1.

Table 1. The proportion of minimal automata in $\mathcal{S}(T_a)$ and $\mathcal{S}(T_b)$

Number of States	21	51	101	201	301	401	499
$\mathbb{P}(\mathcal{M}\|\mathcal{S}(T_a))$	0.5491	0.4461	0.3761	0.3161	0.2853	0.2640	0.2512
$\mathbb{P}(\mathcal{M}\|\mathcal{S}(T_b))$	0.6527	0.5449	0.4620	0.3923	0.3549	0.3289	0.3153

Another aspect to consider is the result of Bassino et al [6], that the asymptotic density of minimal automata for an alphabet of size 2 is about 85%. Our result shows that studying automata from the point of view of their spanning subtree may lead to different behavior than the set of all DFA's. It is also shown by the authors [4], that there are trees for which minimality is generic in $\mathcal{S}(T)$, that is $\mathbb{P}(\mathcal{M}|\mathcal{S}(T)) \to 1$ as $n \to \infty$.

References

1. Almeida, M., Moreira, N., Reis, R.: Enumeration and generation with a string automata representation. Theoretical Computer Science 387(2), 93–102 (2007)
2. Babaali, P.: Generating random automata. In: Proceedings of the 2009 International Conference on Scientific Computing, pp. 85–89 (2009)
3. Babaali, P., Carta-Geradino, E., Knaplund, C.: The number of DFAs produced by a given spanning tree. In: Proceedings of the 2011 International Conference on Scientific Computing, pp. 212–217 (2011)
4. Babaali, P., Knaplund, C.: On the construction of a family of automata that are generically minimal (2012) (preprint)
5. Bassino, F., Nicaud, C.: Enumeration and random generation of accessible automata. Theoretical Computer Science 381(1-3), 86–104 (2007)
6. Bassino, F., David, J., Sportiello, A.: Asymptotic enumeration of minimal automata. In: STACS, pp. 88–99 (2012)
7. Bassino, F., Nicaud, C.: Accessible and Deterministic Automata: Enumeration and Boltzmann Samplers. In: International Colloquium on Mathematics and Computer Science 2006. Discrete Mathematics and Theoretical Computer Science Proceedings, vol. AG, pp. 151–160 (2006)
8. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Proc. Sympos. Math. Theory of Automata (New York, 1962), pp. 529–561. Polytechnic Press of Polytechnic Inst. of Brooklyn, Brooklyn (1963)
9. Champarnaud, J.-M., Paranthoën, T.: Random generation of DFAs. Theoretical Computer Science 330, 221–235 (2005)
10. Domaratzki, M., Kisman, D., Shallit, J.: On the number of distinct languages accepted by finite automata with n states. Journal of Automata, Languages and Combinatorics 7(4), 469–486 (2002)
11. Flajolet, P., Zimmermann, P., Van Cutsem, B.: A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science 132, 1–35 (1994)
12. Harary, F., Palmer, E.M.: Graphical enumeration. Academic Press, New York (1973)
13. Harrison, M.A.: A census of finite automata. In: Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design, pp. 44–46 (1964)
14. Hopcroft, J.: An *nlogn* algorithm for minimizing the states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computation (Proc. Internat. Sympos. Technion, Haifa), pp. 189–196 (1971)
15. Korshunov, A.D.: On the number of non-isomorphic strongly connected finite automata. J. Inf. Process. Cybern. 22, 459–462 (1986)
16. Mäkinen, E.: Generating random binary trees - a survey. Information Sciences: an International Journal 115(1-4), 123–136 (1999)
17. Moore, E.F.: Gedanken experiments on sequential machines. Automata Studies, Annals of Mathematical Studies 34, 129–153 (1956)
18. Nicaud, C.: Étude du compartement en moyenne des automates finis et des langages rationnels. PhD Thesis. University Paris 7 (2000)
19. Nijenhuis, A., Wilf, H.S.: Combinatorial Algorithms: For Computers and Calculators. Academic Press (1978)
20. Robinson, R.W.: Counting strongly connected finite automata. John Wiley & Sons, Inc., New York (1985)

Limited Non-determinism Hierarchy of Counter Automata*

Sebastian Bala and Dariusz Jackowski

Institute of Computer Science
University of Wrocław
Joliot-Curie 20, Wrocław, Poland

Abstract. Automata with weights on edges, especially automata with counters, have been studied extensively in recent years, both because of to their interesting theory and due to their practical applications in data analysis. One of the most significant differences between weighted and classical automata concerns determinization: while every classical automaton can be determinized, this is not the case for weighted automata. Still, obtaining an equivalent automaton as close to a sequential (deterministic) one as possible is crucial in many practical applications, as unbounded non-determinism incurs large computational costs. There exist a few ways to limit the non-determinism of a counter automaton. For each word, one can require that only k runs are accepting (k-ambiguous automata), that there are only k possible runs at all (k-path automata), or one can restrict the automaton itself to be a disjoint sum of k sequential ones (k-sequential automata). Moreover, there are different types of automata with counters: distance automata that cannot reset, desert automata, and R-automata with many counters. In this paper, we establish a hierarchy for all these possibilities. First, we show that the parameter k induces a hierarchy in all cases. Then, we prove that k-path automata can be made 2^{k-1}-sequential and that this bound is strict. Finally, we show an unambiguous automaton which is not finitely sequential at all.

1 Introduction

Automata with the incrementation operation, known as distance automata, where introduced by Hashiguchi in the context of the restricted star height problem. Later, they were extended by Kirsten in [5,6] to nested distance desert automata, which also use the reset operation and have a hierarchical structure. These automata where crucial in a beautiful reduction [6] of the restricted star height problem to the limitedness problem of nested distance desert automata, which resulted in the famous first elementary algorithm for star height. After this success, the interest in counter automata only increased. Automata with many counters and both reset and incrementation operations appeared as ωBS-automata [3] in the context of a decidable extension of the monadic second-order logic with bounding and unbounding quantifiers. Colcombet developed a

* Research supported by the Polish Ministry of Science and Higher Education under grant N N206 492638 2010-2012.

theory of regular cost functions over finite and infinite words [4] representable, among other models, by B-automata and S-automata. Krcal studied in [1] another model, called R-automata, as a specification language for systems operating on finite numbers of resources. The resources are consumed by the systems in small parts, described by incrementation transitions, which can be replenished by actions represented by resetting transitions.

In parallel with the developments in theory described above, weighted automata gained importance in practical applications. Variants of counter automata are used as a data structure in speech recognition and in machine translators [9,10]. There, roughly speaking, the weight of a run on a word represent its cost, and the run with minimal weight corresponds to the optimal answer. Since these automata get very large, finding the optimal run in a general non-deterministic automaton incurs an excessive computational cost in these domains. Of course, that cost is negligible for sequential automata, as then there is just one run for each word. Since counter automata cannot be determinized in general, the research focused on heuristics to decrease the search costs [2] and on sub-classes of automata with limited non-determinism [8,7].

The best known of these sub-classes are *unambiguous* automata, in which there is at most one accepting run on every word. More generally, one can consider the class of *k-ambiguous automata*, where there are at most k accepting runs on each word. Clearly 1-ambiguous is then just another name for unambiguous. The classes of sequential, unambiguous and unrestricted automata are known to be distinct in every studied model of counter automata, see [8] for a more thorough map. For distance automata, Weber also established that the classes of k-ambiguous automata form a strict hierarchy [12,13]. But k-ambiguity does not directly yield any practical benefits. Instead of limiting the number of accepting runs, one would prefer a stronger constraint: allowing at most k runs on every word, whether accepting or not. We use the name *k-path automata* for this class. Note that one needs to trace at most k runs in parallel when using a k-path automaton, a property that can be directly exploited in practice. Finally, the strongest requirement short of determinism is when the automaton is a disjoint sum of k sequential ones. Such *k-sequential* automata can be implemented on k independent machines resulting in fast parallel execution.

With the above-mentioned three models of counter automata (desert, distance, and R-automata) and the three constraints on determinism (k-ambiguous, k-path and k-sequential automata), there is a natural question about their relationship and the hierarchies induced by k. We first show that the parameter k indeed induces a hierarchy, for each of the automata models and each constraint class. This generalizes the results of Weber [12,13] mentioned before. Next, we show that every k-path automaton can be translated to an equivalent 2^{k-1}-sequential one, in each of the models. Moreover, this bound is strict, i.e. we demonstrate a k-path automaton which is not $2^{k-1} - 1$-sequential (again, in all models). Finally, we show an automaton which is unambiguous, and even satisfies an additional stronger constraint, but is not equivalent to any k-path or k-sequential automaton, no matter for which k. The results can thus be summarized as follows. For all of distance, desert and R-automata:

- k-ambiguous, k-path, and k-sequential classes form a strict hierarchy,
- k-path automata can be made 2^{k-1}-sequential and this bound is strict,
- but even basic unambiguous automata are stronger than all k-path ones.

2 Definitions

Let $\mathbb{N}^+ = \mathbb{N}\backslash\{0\}$. For $k \in \mathbb{N}^+$, we will write $[k]$ to denote the set $\{1, 2, \ldots, k\}$.

An R-automaton is a tuple $\mathcal{A} = \langle \Sigma, Q, I, F, \Gamma, \delta\rangle$, where Σ is a finite *input alphabet*, Q is a finite set of *states*, $I \subseteq Q$ is the set of *initial states*, F is the set of *final states*, and Γ is a finite set of *counters*. The *transition relation* δ satisfies

$$\delta \subseteq Q \times \Sigma \times Q \times \Delta^{\Gamma},$$

where $\Delta = \{\mathrm{i}, \mathrm{r}, \epsilon\}$ are the names of operations applied to counters: increment, reset, and leaving them intact. We call $\mathcal{A}$ a *distance automaton* when $|\Gamma| = 1$ and $\delta \subseteq Q \times \Sigma \times Q \times \{\mathrm{i}, \epsilon\}$, i.e. if there is only one counter and it can only be incremented or left intact. $\mathcal{A}$ is a *desert automaton* when $|\Gamma| = 1$ and $\delta \subseteq Q \times \Sigma \times Q \times \{\mathrm{i}, \mathrm{r}\}$, i.e. if there is one counter which, in each step, must be either incremeted or reset.

A *transition* t of an R-automaton $\mathcal{A}$ is a quadruple $t = (p, a, q, \gamma) \in \delta$. A path π in $\mathcal{A}$ of length k is a sequence of transitions $t_1 t_2 \ldots t_k$ such that, for some states $q_0 \ldots q_k$, input letters $a_1 \ldots a_k$ and operations $\gamma_1 \ldots \gamma_k$, it holds that each $t_i = (q_{i-1}, a_i, q_i, \gamma_i)$. The sequence of input letters $a_1 \ldots a_k$ is called the *label* of π and is sometimes denoted by $\mathrm{label}(\pi)$. We write $\pi(i, j)$ to denote the *sub-path* $t_i t_{i+1} \ldots t_j$ of π. For another path π', we will write $\pi' \sqsubseteq \pi$ if it is a sub-path of π, i.e. if $\pi' = \pi(i, j)$ for some i and j. We allow $j = i - 1$, in which case the sub-path is *empty*, in all other cases it is *non-empty*. A subpath $\pi(i, j) = t_i \ldots t_j$ is a *loop* if the starting state of t_i equals the ending state of t_j.

We say that a path $t_1 t_2 \ldots t_k$ in $\mathcal{A}$ is *accepting* if the first state t_1 is initial and the last state t_k is final, i.e. if $t_1 \in I$ and $t_k \in F$. The automaton $\mathcal{A}$ *accepts* a word $w \in \Sigma^*$ if there exists an accepting path π with $\mathrm{label}(\pi) = w$. The automaton $\mathcal{A}$ *recognizes* the language $L(\mathcal{A}) = \{w \mid \mathcal{A} \text{ accepts } w\}$. Note that $L(\mathcal{A})$ does not depend on the counter operations at all.

For a transition $t = (p, a, q, \gamma)$ and a counter $\alpha \in \Gamma$, we will write $v(\alpha, t) = \gamma(\alpha)$ to denote the operation that t effects on the counter α. If there is only one counter, we will slightly abuse this notation and write just $v(t)$. $v(t_1 \ldots t_n, \alpha) = v(t_1, \alpha) \cdots v(t_n, \alpha) \in \Delta^*$.

The valuation of the counters if a vector $\mathbb{N}^{\Gamma}$. This valuation changes according to the semantics of the symbols $\mathrm{i}, \mathrm{r}, \epsilon$. The symbol i stands for *incrementation*, r is a reset, meaning that the counter is re-assigned to 0, and ϵ is called a *pause* and it means that the counter is left unchanged. Formally, for a counter valuation $c \in \mathbb{N}^{\Gamma}$ and a vector of counter operations $op \in \Delta^{\Gamma}$, the result of *applying op* on c is a valuation $c' \in \mathbb{N}^{\Gamma}$ such that $c'(\alpha) = c(\alpha) + 1$ if $op(\alpha) = \mathrm{i}$, $c'(\alpha) = 0$ if $op(\alpha) = \mathrm{r}$, and $c'(\alpha) = c(\alpha)$ if $op(\alpha) = \epsilon$.

For a given R-automaton $\mathcal{A}$ and a path $\pi = t_1 \dots t_k$, let $c_0 \dots c_k$ be the sequence of counter valuations such that $c_0 = 0^{\Gamma}$ and c_{i+1} is the result of applying γ_i on c_i, where γ_i is the counter operation of the transition t_i. Let $d(\pi)$ be the maximal value reached by a counter during the run along π, i.e. $d(\pi) = \max\{c_i(\alpha) \mid i \leq k, \alpha \in \Gamma\}$. We define $d_{\mathcal{A}} : \Sigma^* \to \mathbb{N} \cup \infty$ as

$$d_{\mathcal{A}}(w) = \begin{cases} \min\{d(\pi) : \pi \text{ is labeled by } w \text{ and accepting}\} & \text{if } w \in L(\mathcal{A}), \\ \infty & \text{otherwise.} \end{cases}$$

Two R-automata $\mathcal{A}$ and $\mathcal{B}$ are *equivalent* if $d_{\mathcal{A}} = d_{\mathcal{B}}$.

For an R-automaton $\mathcal{A}$, let us define two functions, $amb_{\mathcal{A}}, tamb_{\mathcal{A}} : \Sigma^* \to \mathbb{N}$, which return, for a word w, respectively the number of accepting paths labeled by w and the number of all paths labeled by w in $\mathcal{A}$ (not necessarily accepting).

We say that that $\mathcal{A}$ is k-ambiguous, or k-amb, if $amb_{\mathcal{A}}(w) \leq k$ for all $w \in L(\mathcal{A})$. An automaton is *complete* if for all states $q \in Q$ and all letters $a \in \Sigma$ there exists transition (q, a, p, γ) to a state $p \in Q$. This simple condition is necessary to reasonably account for all non-accepting runs, as will become clear later. We call $\mathcal{A}$ a k-path automaton, or just k-path, if it is complete and $tamb_{\mathcal{A}}(w) \leq k$ for all $w \in L(\mathcal{A})$.

An automaton $\mathcal{A}$ is sequential, or deterministic, if for every state $q \in Q$ and letter $a \in \Sigma$ there exists at most one transition $(q, a, p, \gamma) \in \delta$, and if $|I| = 1$, i.e. there is exactly one initial state. We call $\mathcal{A}$ finitely sequential if it satisfies the first condition, but not the second one, i.e. I may be arbitrary. We say that it is k-sequential, or k-seq, if $|I| \leq k$.

Slightly abusing notation, we will use the terms k-seq, k-path, and k-amb to denote both the classes of automata $\mathcal{A}$ as well as the classes corresponding functions $d_{\mathcal{A}}$. It follows directly from the above definitions that the following sequence of inclusions is satisfied.

$$k - \mathsf{seq} \subseteq k - \mathsf{path} \subseteq k - \mathsf{amb}.$$

Let $\mathcal{A}_i = \langle \Sigma, Q_i, I_i, F_i, \Gamma, \delta_i \rangle$ (for $i = 1, \dots, s$) be R-automata with pairwise disjoint sets of states. The *disjoint union* of $A_1, \dots, A_s$, denoted by $\biguplus_{i=1}^{s} A_i$, is an R-automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \Gamma, \delta \rangle$ where

$$Q = \bigcup_{i=1}^{s} Q_i, \quad I = \bigcup_{i=1}^{s} I_i, \quad F = \bigcup_{i=1}^{s} F_i, \quad \delta = \bigcup_{i=1}^{s} \delta_i.$$

Note that every k-sequential automaton $\mathcal{A}$ with $I = \{i_i, \dots, i_k\}$ can be represented as a disjoint union of k *deterministic components* $\mathcal{A}_1, \dots, \mathcal{A}_k$. Each component $\mathcal{A}_j$ is equal to $\mathcal{A}$ except that it has $I = \{i_j\}$.

3 Hierarchies Induced by the Parameter *k*

In this section, we show that the parameter k induces a strict hierarchy of k-amb, k-path, and k-seq automata, for distance, desert, and the general R-automata as well. Among the results presented in [13] it has been proved that $k - \mathsf{amb} \subsetneq (k+1) - \mathsf{amb}$ for distance automata, and the constructions in those proofs imply that $k - \mathsf{amb} \not\subseteq (k+1) - \mathsf{seq}$ in that model. The following theorem uses a similar construction to generalize the result to other models.

Theorem 1. *For every $k \in \mathbb{N}^+$, there exists $(k+1)$-seq distance (desert) automaton which isn't equivalent to any k-amb R-automaton.*

Before we start the proof, let us remark that in [11] Sakarovitch and de Souza proved that k-amb $\mathbb{N}$-weighted automata can be translated into equivalent automata which are a disjoint union of k unambiguous automata. Their technique, multi-skimming, i.e. covering of an $\mathbb{N}$-weighted automaton, works in particular for R-automata. We exploit this fact in the form of the following corollary of their result.

Corollary 2.. *Let $k \in \mathbb{N}$. Let $\mathcal{M} = \langle \Sigma, S, Q, F, \Gamma, \delta \rangle$ be a k-amb R-automaton. Then, there exist k unambiguous R-automata $\mathcal{M}_1, \ldots, \mathcal{M}_k$ such that M is equivalent to disjoint union of $\mathcal{M}_1, \ldots, \mathcal{M}_k$.*

Proof. Let $\Sigma = \{a_0, \ldots, a_k\}$. Let $d_{\mathcal{A}} : \Sigma^* \mapsto \mathbb{N}$ be a function which assigns to every string $w \in \Sigma^*$ the length of the shortest subword u such that $u = (a_i)^l$ for some $i \in [k]$ and $l \geq 1$, and such that $w = w_1 u w_2$ for some $w_1, w_2 \in (\Sigma \setminus \{a_i\})^*$. For strings which shuffle one letter with another the value of $d_{\mathcal{A}}$ equals ∞. It's easy to construct $(k+1)$-seq desert automaton $\mathcal{A} = \langle \Sigma, S, Q, F, \Gamma, \delta \rangle$ recognizing Σ^* with weights determined by $d_{\mathcal{A}}$: $Q = F = S = \{q_0, \ldots, q_k\}$ and $\gamma = 1$ for $(q_i, a_j, q_i, \gamma) \in \delta$ if $i = j$, 0 otherwise. Hence the function $d_{\mathcal{A}} \in (k+1)$-$\mathsf{seq}$.

Now the proof goes as follows: suppose that there exists a k-amb automaton $\mathcal{M}' = \langle \Sigma, S', Q', \Gamma', F', d' \rangle$ equivalent to $\mathcal{A}$. From Corollary 2, $\mathcal{M}'$ can be decomposed into k unambiguous R-automata $\mathcal{M}_1$, $\mathcal{M}_2$,... $\mathcal{M}_k$. For each $\mathcal{M}_i$ it is easy to build an unambiguous automaton $\mathcal{M}'_i$ such that $L(\mathcal{M}'_i) = \Sigma^* \setminus L(\mathcal{M}_i)$ and all of its transitions are of the type $i\epsilon^{|\Gamma|-1}$. Let $\tilde{\mathcal{M}}_i = \mathcal{M}_i \uplus \mathcal{M}'_i$. The automaton $\tilde{\mathcal{M}}_i$ is unambiguous because both components $\mathcal{M}_i$ and $\mathcal{M}'_i$ are unambiguous and, moreover, they recognize disjoint languages.

Note that distance automata $\mathcal{A} = \biguplus_{i=1}^{k} \mathcal{M}_i$ and $\tilde{\mathcal{A}} = \langle \Sigma, \tilde{S}, \tilde{Q}, \tilde{F}, \tilde{d} \rangle$, where $\tilde{\mathcal{A}} = \biguplus_{i=1}^{k} \tilde{\mathcal{M}}_i$, are equivalent. This holds because both these automata recognize Σ^* and the weight of a string w in $\mathcal{A}$ is the same as in $\tilde{\mathcal{A}}$, for all $w \in \Sigma^*$. The weights of w in $\mathcal{A}$ and $\tilde{\mathcal{A}}$ are equal because all accepting paths in $\mathcal{A}$ have counterparts in $\tilde{\mathcal{A}}$ and all accepting paths in $\tilde{\mathcal{A}}$ with no counterparts in $\mathcal{A}$ come from the component M'_i (for some i). Their weights are equal of their length, because of the incrementation of the first counter in each transition.

Let n be the maximum number of states of the automata $\tilde{\mathcal{M}}_1$, $\tilde{\mathcal{M}}_2$,... $\tilde{\mathcal{M}}_k$ and let $w = a_0^{n^k} a_1^{n^k} \ldots a_k^{n^k}$. Let π_i be a path in $\tilde{\mathcal{M}}_i$ accepting w and let $\pi_{i,j}$ be a subpath of π_i corresponding to $a_j^{n^k}$. Then there exist $l_j^{(1)}$, $l_j^{(3)} \in \mathbb{N}$ and $l_j^{(2)} \in \mathbb{N}^+$ such that $l_j^{(1)} + l_j^{(2)} + l_j^{(3)} = n^k$ and, for each i, $\pi_{i,j}$ can be factorized into $\pi_{i,j}^{(1)}$, $\pi_{i,j}^{(2)}$, $\pi_{i,j}^{(3)}$. The subpath $\pi_{i,j}^{(m)}$ consumes $a_j^{l_j^{(m)}}$ and, for each i and j, $\pi_{i,j}^{(2)}$ is a loop (i.e. it has the same starting and ending state).

First, note that weight of each path π_i is at least n^k because the weight of a string w equals n^k. All loops $\pi^{(2)}_{i,j}$ have lengths at most n^k. We call a loop $\pi^{(2)}_{i,j}$ *growing* if there exists a counter which is incremented and not reset any time along $\pi^{(2)}_{i,j}$. Otherwise, we call a loop *non-growing.*

Note that multiple iterations of a loop $\pi^{(2)}_{i,j}$, which is non-growing, increase the weight of the subpath $\pi^{(2)}_i$ at most by $|\pi^{(2)}_i| - 1$[1] independently of the number of iterations. Otherwise, when the iterated loop is growing, the weight of the outcome path increases at least by $l_{i,j}$, where $l_{i,j}$ is the number of iterations that have been applied to $\pi^{(2)}_{i,j}$.

Note that, for every $\alpha \in \{0, \dots, k\}$, denoting the part of w containing $a_\alpha^{n^k}$, there exists $i = i(\alpha)$, indexing the number of component $\tilde{M}_i$ and i^{th} accepting path labeled by w, such that, for every $j \in \{0, 1, \dots k\}$, if $\alpha \neq j$, then the path $\pi^{(2)}_{i,j}$ is non-growing. Otherwise, there may exist α such that for all accepting paths π_i labeled by w one can find $j(i) \neq \alpha$ such that weight of $\pi^{(2)}_{i,j(i)}$ is growing. In such case all loops $\pi^{(2)}_{i,j(i)}$ could be pumped up n^k+1 times, in the parts $\pi^{(2)}_{i,j(i)}$, producing k accepting paths of weight greater than n^k labeled by pumped word

$$w' = a_0^{n^k+l_0^{(2)}(n^k+1)} \dots a_{\alpha-1}^{n^k+l_{\alpha-1}^{(2)}(n^k+1)} a_\alpha^{n^k} a_{\alpha+1}^{n^k+l_{\alpha+1}^{(2)}(n^k+1)} \dots a_k^{n^k+l_k^{(2)}(n^k+1)}$$

of weight exactly n^k. This contradicts the fact that $\tilde{A}$ and A are equivalent.

Now let α, α_1 be such that $\alpha \neq \alpha_1$ and $i_0 = i(\alpha) = i(\alpha_1)$. Such pair exists by the pigeon hole principle. Then, all subpaths $\pi^{(2)}_{i_0,j}$ are non-growing. Let $t \geq n^k$ be a weight of the path π_{i_0} and $c = (k+1) \cdot \max\{l_0^{(2)}, \dots, l_k^{(2)}\} + \text{t}$. Consider the word $w' = a_0^{n^k+l_0^{(2)}c} a_1^{n^k+l_1^{(2)}c} \dots a_k^{n^k+l_k^{(2)}c}$. By previous observation – that iterations of non-growing loops do not increase the weight of the path less than by the length of the loop – we obtain that π_{i_0}, after pumping, is transformed into an accepting path π'_{i_0}. It labels w' with the weight $d(\pi'_{i_0})$ not greater than $t + \sum_{i=0}^k l_i^{(2)} \leq c$. According to the equivalence of $\tilde{\mathcal{A}}$ and $\mathcal{A}$, the weight of π'_{i_0} should be at least $n^k + c$. Since $n^k + c > t$, this is a contradiction. Therefore $\tilde{\mathcal{A}}$ does not exist and, consequently, neither does M'. $\square$

4 Finitely Sequential and *k*-Path Automata

In this section, we show that k-path automata are 2^{k-1}-sequential and that this bound is tight. We will start with the counterexample that proves the tightness of the bound. To make uniform proofs for distance and desert automata, let us first define an auxiliary function θ, such that $\theta : \{0,1\} \to \{\epsilon, \text{i}\}$ for distance automata and $\theta : \{0,1\} \to \{\text{r}, \text{i}\}$ for desert automata.

[1] An even better estimation can be done – $\left\lfloor \frac{|\pi_i^{(2)}|}{2} \right\rfloor$

$$\theta(x) = \begin{cases} i & \text{if } x = 1, \\ \epsilon^{\text{ (for distance aut.)}} \mid r^{\text{ (for desert aut.)}} & \text{otherwise.} \end{cases}$$

Now let us define the automaton $\mathcal{A} = \langle \Sigma, Q, I, F, \Gamma, \delta \rangle$ which is a k-path distance (or, desert depending on the choice in the function θ) automaton. It will be easy to see from its definition that it belongs to the k-path class, and we will use it as a counterexample in the next proof.

Let $\Sigma = \{a, 0, 1\}$, $S = \{f_{1,1}\}$, $F = \{f_{j,l} | j \in \{0,1\},\ l \in [k]\} \setminus \{f_{0,1}\}$, $Q = F \cup \{[gb]\}$, and define the distance function as follows

- $v(f_{j,l}, a, f_{j,l}) = \theta(j)$;
- $v(f_{j,l}, x, [gb]) = \theta(0)$ if $j \neq x$ and $x \in \{0,1\}$ or $l = k$;
- $v([gb], x, [gb]) = \theta(0)$, where $x \in \Sigma$;
- $v(f_{x,l}, x, f_{y,l+1}) = \theta(y)$ if $l < k$, $x, y \in \{0,1\}$;

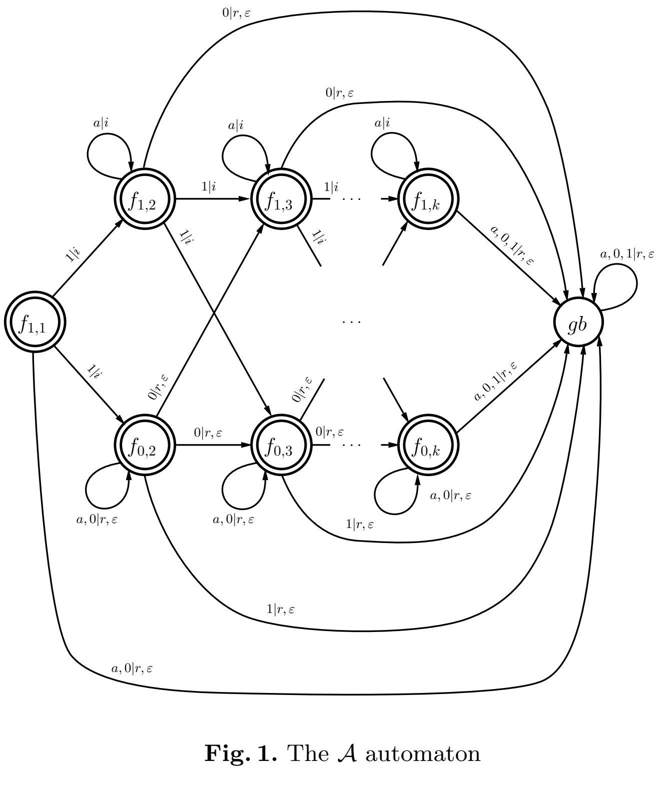


Fig. 1. The $\mathcal{A}$ automaton

The automaton $\mathcal{A}$ accepts all $0-1$ sequences $b_1, \ldots, b_s$ separated by sequences of letter a, such that $s \leq k$. In the distance version, the automaton counts occurrences of 1 together with a letters standing right before them. In the desert version, it counts the length of the longest sequence of a standing right before a 1 occurrence plus one. The automaton $\mathcal{A}$ is k-path because one can chose an arbitrary state $q \in Q$ and for all letters $x \in \Sigma$ there exists a transition starting at q and labeled by x. The following theorem is proved in Appendix A.

Theorem 3. *For every $k \in \mathbb{N}^+$, there exists a k-*path *desert (distance) automaton which is not equivalent to any $(2^{k-1}-1)$-*seq *R-automaton.*

We will now show a method of translating k-path distance, desert and R-automata to an equivalent 2^{k-1}-seq automata of the same type. Let S be a set which contains finite number of tokens. Let T_k be an arbitrary finite tree with k leaves. The pair (S, T_k) represents a *process of uniform token distribution* which works as follows:

(1) The distribution starts at the root of T_k, where we put all tokens.
(2) At every step of the distribution, all tokens placed at some inner node or are moved to its children.
(3) The tokens are moved in such a way that the difference between the numbers of tokens at each pair of children is at most one, after replacement.
(4) The process ends when all tokens are distributed to the leaves of T_k.

Note that distribution process is not necessarily unambiguous. Many different traces of distribution satisfy the definition of uniform token distribution.

Lemma 4. *Let T be a tree with k leaves and let S be a set which contains $2^{(k-1)}$ tokens. Then an (S, T) process of uniform token distribution always ends with the configuration where every leaf gets at least one token.*

Proof. First consider how many tokens each vertex needs to satisfy the lemma's condition in its subtree. Clearly each leaf needs only one token and each internal vertex needs the maximum number of tokens required for one of it's children multiplied by number of its children. Therefore the function f which returns, for each inner node v of T, the minimal number of tokens required to cover the leaves of the subtree of T rooted in v after the process, can be defined in the following way. Let $C(v)$ be the set of children of node v.

$$f(v) = \begin{cases} 1 & \text{if v is a leaf ,} \\ |C(v)| * \max_{w \in C(v)} f(w) & \text{otherwise} \end{cases}$$

The minimum tokens number needed to cover the leaves of T is $f(root(T))$ and will be denoted by $f(T)$. From now to the end of this proof, a path p of a tree T denotes the set of node labels occurring in some path from the root of T to some leaf of T. We assume that each node in a tree has a unique label.

Fact 5. *In the tree T there exists the path p from root to one of the leaves such that $f(T) = \Pi_{v \in p'} C(v)$, where p' is p without ending leaf.*

Proof. The proof is by induction on the height of T. If the height of T is 1, it's obvious, as $f(T) = 1$ and, since the root of T is also a leaf, there is only one path in T for which the product from this fact is 1 (because p' is empty). Now suppose that this fact holds for every tree with height less than n. Let the height of T be n and let v be a child of $root(T)$ such that $f(v) = \max_{w \in C(root(T))} f(w)$, let p be a path in subtree rooted in v such that $f(v) = \Pi_{w \in p'} C(w)$, where p' is p without the ending leaf – such p exists by inductive assumption. Now $f(T) = |C(root(T))| * f(v) = |C(root(T))| * \Pi_{w \in p'} C(w) = \Pi_{w \in p''} C(w)$, where p'' is p' with $root(T)$ attached at the beginning. □

Now we are ready to prove Lemma 4. Let T be a tree with k leaves and p, p' be the paths from Fact 5. We show that $\Pi_{w \in p'} C(w) \leq 2^{k-1}$ by the following induction over the number of vertices in p'.

– If $p' = 0$ then T has only one vertex, $k = 1$ and $f(T) = 1 = 2^0$.
– Let p' be a path such that $f(T) = \Pi_{v \in p'} C(v)$. Let p'' be p' without the root of T, and $n = |C(root(T))|$, and $d = \Pi_{w \in p''} C(w)$. Let $T^1, \ldots, T^n$ be trees rooted at children of $root(T)$. $f(T^i) \leq d$ for all $1 \leq i \leq n$ and $f(T^i) = d$ for at least one i. Note that $f(T) = nd$. Since all trees T^i contain at least one leaf of the tree T then, by induction hypothesis $d \leq 2^{k-1-(n-1)}$ for all i. Therefore $f(T) = nd \leq n2^{k-1-(n-1)} \leq 2^{k-1}$. □

Theorem 6. *For every k-path $n-$state R-automaton (distance, desert) there exists an equivalent 2^{k-1}-seq R-automaton (distance, desert) with $2^{3k-3}n$ states, of which 2^{k-1} are starting ones.*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, I, F, \Gamma, \delta \rangle$ be a k-path R-automaton (distance, desert) with $n-$states. We define an equivalent $2^{(k-1)}$-seq automaton $\mathcal{B} = \langle \Sigma, Q', I', F', \Gamma, \delta' \rangle$ as follows. Let $Q' = \{(i, p, b, e) : 1 \leq i \leq 2^{k-1}, p \in Q,\ 1 \leq b \leq e \leq 2^{k-1}\}$ where i is a marker which assigns the state to particular deterministic component of automaton $\mathcal{B}$. Assume that tokens from Lemma 4 are linearly ordered and indexed by numbers $1, \ldots, 2^{(k-1)}$. Let $D[b, e, n]$ denote uniform division of interval of token indexes $[b, e]$ on n adjacent disjoint subintervals. Difference between the number of elements of any pair of intervals in $D[b, e, n]$ is at most one. Assume that consecutive intervals at $D[b, e, n]$ are created according to nondecreasing number of elements. Let $D[b, e, n][i]$ denotes i^{th} interval in the ordering. Also assume that the set of states Q is linearly ordered. Let $S[j]$ denotes j^{th} element of S with respect to the ordering. Define S' as

$$\Big\{(i, p, b, e) \mid 1 \leq b \leq i \leq e \leq 2^{k-1},\ [b, e] = D[1, 2^{(k-1)}][j] \text{ and}$$

$$p = S[j] \text{ for } j \in [|S|]\Big\},$$

$F' = \{(i, p, b, e) \in Q' \mid p \in F\}$. Let $C(p, a) = \{t \in Q \mid d(p, a, t) \neq \infty\}$. By $C(p, a)[j]$ denote $j-$th state in $C(p, a)$ with respect to the ordering over set of states.

Let us list relations between the components of the new states, that we need to define δ' :

$$[bb, ee] = D[b, e, n][k] \tag{1}$$

$$q = C(p, a)[k] \tag{2}$$

$$n = |C(p, a)| \tag{3}$$

$$bb \leq i \leq ee. \tag{4}$$

Now δ' for $\mathcal{B}$ is defined as

$$\{((i,p,b,e),a,(i,q,bb,ee),\gamma)\,|(p,a,q,\gamma) \text{ if for some } k \in [n], \}$$

The intuition behind this construction goes as follows. By the definition of k-path automata, the number of paths for any w in $\mathcal{A}$ is not greater than k. Moreover, it cannot happen that some of the nondeterministic paths disappear on a proper prefix of w. Then the paths form a 'solid' tree T_w with paths stretched from the beginning of w to the last letter of w. Tree T_w has at most k leaves for any w. The root of this tree is a dummy node with children that correspond to all states from S. One can think that, for an arbitrary string w, tokens are uniformly spread out to cover all nondeterministic choices of the automata $\mathcal{A}$ working over w. By Lemma 4, the process of uniform token distribution puts finally some token at leaves of an arbitrary tree with k leaves when it starts with $2^{(k-1)}$ tokens. Each component of $\mathcal{B}$ corresponds to one token. A particular component inherits from $\mathcal{A}$ these paths which contain a corresponding token from the beginning to the end of the token distribution process. The token corresponding to any component of $\mathcal{B}$ is a part of a state description of all states. Uniform distribution, which spread tokens at each node with nondecreasing number of tokens looking from the left to the right son, is unambiguous (we fix order of nondeterministic choices at each step of computation). This fact determine that outcome of $\mathcal{B}$ construction is a function δ'. Hence $\mathcal{B}$ is a finitely sequential automaton. Since $\mathcal{B}$ has $2^{(k-1)}$ components each path of automaton $\mathcal{A}$ working over arbitrary w has his counterpart at some component of $\mathcal{B}$, and automaton $\mathcal{A}$ is equivalent to $\mathcal{B}$. $\square$

5 The Power of Unambiguous Automata

In this section, we prove the following theorem by showing a counterexample.

Theorem 7. *There exists an unambiguous distance (desert) automaton which is not a j-seq R-automaton for any $j \in \mathbb{N}^+$. It is thus not equivalent to any j-seq R-automaton for any $j \in \mathbb{N}^+$.*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, I, F, \Gamma = \{\alpha\}, \delta \rangle$ be a distance or desert automaton, where: $\Sigma = \{a,b,c\}$, $Q = \{p,q\}$, $S = \{p\}$, $F = \{q\}$, and

$$\delta = \{(p,a,p,\theta(1)),(p,c,p,\theta(1)),(p,a,q,\theta(0)),(q,b,q,\theta(0)),(q,c,q,\theta(0))\},$$

where θ is the function defined in the previous section. Recall that the definition of θ depends on which type of automata is considered, whether distance or desert. It is easy to notice that $\mathcal{A}$ is unambiguous.

For the sake of contradiction, assume that there exists a finitely sequential R-automaton $\mathcal{B} = \langle \Sigma, Q', I', F', \Gamma', \delta' \rangle$ equivalent to $\mathcal{A}$. Let k denote number of starting states of $\mathcal{B}$. Let $c_i = c^{2^{i+2}-1}a$, $b_i = c^{2^{i+2}-1}b$, $r_{-1} = a$, $r_i = r_{i-1}c_{k-i}$, $s_i = r_{i-1}b_{k-i}$.

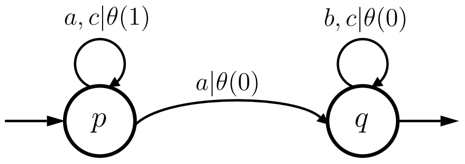


Fig. 2. Illustration for the Proof of Theorem 7

We show that each of word $s_0, s_1, \dots s_k$ is accepted with minimum weight by a different deterministic component of $\mathcal{B}$. For s_0 and s_1 we have $d_{\mathcal{A}}(s_0) = d_{\mathcal{A}}(ab_k) = 0$ and $d_{\mathcal{A}}(s_1) = d_{\mathcal{A}}(ac_k b_{k-1}) = 2^{k+2}$. Now suppose that these two words are accepted by the same deterministic component in $\mathcal{B}$. In this component, for the prefix $ac^{2^{k+2}-1}$, there cannot be any transition t such that $v(t, \alpha) = i$ for some $\alpha \in \Gamma'$ (because $d_{\mathcal{A}}(s_0) = 0$). Otherwise, there could exist a counter in Γ' which gets positive value in the accepting path labeled by s_0 and, by the assumption that the path was chosen with minimal weight, $d_{\mathcal{B}}(s_0) > 0$. Hence, if a transition $t = (q_1, x, q_2, \gamma)$ comes from the the proper prefix of s_0, then $\gamma \in (\Delta \setminus \{i\})^{|\Gamma'|}$. But this means that the weight of s_1 can be derived from suffix ab_{k-1}, which is impossible because $|ab_{k-1}| = 2^{k+1} + 1 < 2^{k+2}$. Before we proceed, consider the following two observations:

Observation 8. $d_{\mathcal{A}}(s_i) = |s_{i-1}|$.

Observation 9. *In every accepting path π_{si} labeled by s_i, with minimal weight, the path π_{rj} which is the prefix of π_{si} labeled by r_j, such that $1 < j < i$, it holds that $d(\pi_{rj}) \geq |r_{j-1}| + 2$.*

Assuming that $d(\pi_{rj}) < |r_{j-1}| + 2$, we obtain that no counter reaches the value $|r_{j-1}| + 2$ in π_{rj}. Let $\pi_{si} = \pi_{rj}\pi$. The length of π is less than 2^{k-j+2}. Hence the greatest value reached by some counter is less than

$$|r_{j-1}| + 2 + 2^{k-j+2} \leq 2^{k+2} - 2^{k-j+3} + 1 + 2 + 2^{k-j+2} =$$

$$= 2^{k+2} - 2^{k-j+2} + 3 < |s_{i-1}| = 2^{k+2} - 2^{k-(i-1)+2} + 4.$$

This is a contradiction with Observation 8.

Assume that $j < i$. Suppose that s_i and s_j are accepted with minimal weight by the same deterministic component of $\mathcal{B}$. Strings s_i and s_j have a common prefix of length $|s_j| - 1$. Since s_j is accepted with the minimal weight $|s_{j-1}|$ along the path π_{sj} (this is the value reached by some of the counters), the path π_j, which is a prefix of π, labeled by its prefix of length $|s_j| - 1$, has weight not greater than $|s_{j-1}|$. By Observation 8, the weight of the same path is greater than $|r_{j-1}| + 1$. Since $|s_{j-1}| = |r_{j-1}|$, we obtain a contradiction. □

References

1. Abdulla, P.A., Krcal, P., Yi, W.: R-Automata. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 67–81. Springer, Heidelberg (2008)
2. Allauzen, C., Riley, M.D., 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)
3. Bojanczyk, M., Colcombet, T.: Bounds in w-regularity. In: LICS. pp. 285–296. IEEE Computer Society (2006)
4. Colcombet, T.: The Theory of Stabilisation Monoids and Regular Cost Functions. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 139–150. Springer, Heidelberg (2009)
5. Kirsten, D.: Distance Desert Automata and the Star Height One Problem (Extended Abstract). In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 257–272. Springer, Heidelberg (2004)
6. Kirsten, D.: Distance desert automata and the star height problem. ITA 39(3), 455–509 (2005)
7. Kirsten, D., Lombardy, S.: Deciding unambiguity and sequentiality of polynomially ambiguous min-plus automata. In: Albers, S., Marion, J.Y. (eds.) STACS. LIPIcs, vol. 3, pp. 589–600. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)
8. Klimann, I., Lombardy, S., Mairesse, J., Prieur, C.: Deciding unambiguity and sequentiality from a finitely ambiguous max-plus automaton. Theor. Comput. Sci. 327(3), 349–373 (2004)
9. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23(2), 269–311 (1997)
10. Mohri, M., Pereira, F., Riley, M.: Weighted finite-state transducers in speech recognition. Computer Speech & Language 16(1), 69–88 (2002)
11. Sakarovitch, J., de Souza, R.: On the decomposition of k-valued rational relations. In: Albers, S., Weil, P. (eds.) STACS. LIPIcs, vol. 1, pp. 621–632. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2008)
12. Weber, A.: Distance automata having large finite distance or finite ambiguity. Mathematical Systems Theory 26(2), 169–185 (1993)
13. Weber, A.: Finite-valued distance automata. Theor. Comput. Sci. 134(1), 225–251 (1994)

Unambiguous Automata Denoting Finitely Sequential Functions*

Sebastian Bala and Artur Koniński

Institute of Computer Science
University of Wrocław
Joliot-Curie 20, Wrocław, Poland

Abstract. The min-plus automata with real weights are interesting both in theory and in practice, e.g. their variants are used as a data structure in speech recognition. In this paper we study automata which are finite unions of deterministic ones, called finitely sequential automata. Such automata allow fast detection of optimal paths in parallel while still allowing to express ambiguous functions. We provide a polynomial time algorithm which decides if the given min-plus unambiguous automaton, with rational weights, has a finitely sequential version and we show how to build such equivalent one if the answer is positive. To this end, we introduce the Fork Property which plays the same role as the negation of the Twin Property in case of determinisation. We show that an unambiguous automaton can be transformed into a finitely sequential one if and only if the Fork Property is not satisfied.

1 Introduction

The min-plus automata and transducers have been used successfully to represent models derived from large data sets. In automatic speech recognizers or translators they are used as a data structures encoding possible translations together with probabilities that a translation is proper. One of the tasks of the translation is to find the best or most probable path for the given input. When the automaton is deterministic (sequential) the searching process can be done much faster than in case of fully nondeterministic automata. However, in contrast to the classical automata, weighted automata do not always have a deterministic equivalent representing the same mapping from strings to its values. The determinisation problem asks, given an automaton, whether there exists an equivalent deterministic one. It is still open whether the determinisation problem is decidable and it probably cannot be solved efficiently in general. Hence heuristic algorithms gained importance in speech recognition applications. The heuristics of pruning transitions with substantially smaller weights, approximations [3] and exploiting sufficient conditions for determinisation such as the Twin Property were studied from '90s until recently [1] and some of them have been implemented [11], [2].

* Research supported by the Polish Ministry of Science and Higher Education under grant N N206 492638 2010-2012.

For automata which are a union of deterministic ones, searching for the best solution can be done in parallel. Until the number of deterministic components is not too big, the post-processing which establishes the optimal solution can be advantageous in comparison to searching methods applied usually for nondeterministic automata. One can conjecture that the question if a min-plus automaton over real numbers has a finitely sequential equivalent is not easier than the existence of a deterministic version. Therefore we deal with the problem for less complex inputs such as unambiguous automata. It is known that unambiguous min-plus automata can describe function which cannot be described by any finitely sequential min-plus automaton [8].

This paper provides an algorithm for deciding the existence of a finitely sequential counterpart for a given unambiguous min-plus automaton. The algorithm works in polynomial time by deciding the Fork Property. When the given automaton is unambiguous, the Fork Property can be seen as a generalization of the Twin Property which is a necessary and sufficient condition for the existence of a deterministic counterpart [10]. We prove that the negation of the Fork Property is a necessary and sufficient condition for the existence of a finitely sequential counterpart of an automaton. We also present how to construct the suitable set of deterministic automata if the Fork Property does not hold. Similar properties have been defined for translations of finitely ambiguous [8] and polynomially ambiguous automata [7] into unambigous ones.

Organisation: Section 2 presents basic definitions and a map of decision problems concerning translations between the nondeterminism-level classes of min-plus automata. Section 3 contains the definitions closely related to the proof and next the proof that, for a given unambiguous min-plus automata, the Fork Property is necessary and sufficient for non-existence of a finitely sequential equivalent. In the paragraph 'Complexity' we give an upper bound on the size of the finitely sequential output if the unambiguous input does not satisfy the Fork Property and an analysis of the complexity of establishing whether the Fork Property holds or not.

Related Work. For classical finite automata over finite strings or trees, deterministic automata describe the same class of languages as nondeterministic ones. The situation is quite different for weighted automata. With respect to the forms of nondeterminism, functions recognizable by automata over the tropical semirings form a hierarchy [8,5] which can be depicted as follows:

$$\begin{tikzcd}
 & \mathsf{Fseq} \arrow[dr, "\subsetneq"] & \\
\mathsf{Seq} \subsetneq (\mathsf{Namb} \cap \mathsf{Fseq}) \arrow[ur, "\subsetneq"] \arrow[dr, "\subsetneq"'] & & \mathsf{Famb} \subsetneq \mathsf{Pamb} \subsetneq \mathsf{Rat} \\
 & \mathsf{Namb} \arrow[ur, "\subsetneq"'] &
\end{tikzcd}$$

It is not known in general whether the determinisation problem is decidable. The known decidablity result [7] works for polynomially ambiguous automata, for which it is decidable in a constructive way if a deterministic equivalent exists. One of the results presented in [7] is the proof that the Intermediate Property is a necessary and sufficient condition for polynomially ambiguous automaton to have an unambiguous equivalent. The property says that there is a number Y

such that if a path is accepting and minimal for its label then other accepting paths with the same label cannot deviate, reaching substantially smaller weight on a prefix (with the difference smaller than Y), from the weight of the prefix of the optimal path.

As for the complexity of the decision problem, the equivalence between a polynomially ambiguous automaton and an unambiguous automaton is triply exponential in the number of states of the first one. The equivalence problem between min-plus automata its known to be undecidable in general. When one of given automata is unambiguous it becomes decidable.

For unambiguous automata, the determinisation problem has a positive answer if and only if the Twin Property holds [4,10]. The Twin Property is decidable in polynomial time ($O(|Q|^2 + |E|^2)$, E is the set of transitions) for unambiguous automata [1] and in polynomial space for arbitrary automata [6]. Moreover, the determinisation algorithm [10] returns automata of size at most exponential with respect to the size of the input.

For finitely ambiguous automata [8] the determinization problem has been solved in two stages: First deciding if a translation to an unambiguous automaton is possible and then deciding the Twin Property over the unambiguous equivalent. The first stage starts from translation of the given automaton into a finite union of unambiguous ones. The best known translation is exponential in the size of the automaton [12]. Then the Dominance Property has been defined over the new representation – the union of unambiguous automata. The Dominance Property separates these unions which can be translated into unambiguous automata from those which have no unambiguous equivalent. The property is decidable in polynomial time with respect to the union, hence exponential with respect to the input.

Below we summarize results concerning translations. A row label denotes the source and a column label denotes the destination class of a translation. Each cell contains a description of the time complexity class to which the translation belongs and the translation's size. The complexities presented in the table follow from the proofs presented in the cited papers.

	pamb	famb	fseq	namb	seq
rat	?	?	?	?	?
pamb	-	?	?	*, 3-EXP [7]	*, 4-EXP [7]
famb	-	-	?	EXP, 2-EXP [8]	EXP, 3-EXP [8]
fseq	-	-	-	P, EXP [8]	P, 2-EXP [8]
namb	-	-	P, 2-EXP [here]	-	P, EXP [10]

* The solution of the decision problems presented in [7] and marked by the star symbol in the table above depends on the equivalence problem between polynomially ambiguous and unambiguous automata. This problem is known to be decidable by [9] there is no any detailed estimation of the procedure and we could only speculate about the complexity.

2 Preliminaries

Formally a *weighted finite automaton over a semiring* $\mathbb{K} = \langle K, +, \cdot, 1_K, 0_K\rangle$ (wfa) is a quintuple $\mathcal{A} = \langle Q, \Sigma, \lambda, \mu, \gamma\rangle$ where Σ is a finite alphabet, Q is a finite set called states, $\lambda, \gamma \in \mathbb{K}^Q$, and $\mu : \Sigma^* \to \mathbb{K}^{Q\times Q}$ is a homomorphism into the semiring of $Q\times Q$-matrices over $\mathbb{K}$. A state $q \in Q$ is called *initial* if $\lambda[q] \neq 0_K$ and *final* if $\gamma[q] \neq 0_K$.

A quadruple $(p, a, l, q) \in Q \times \Sigma \times \mathbb{K} \times Q$ is a *transition* of the wfa $\mathcal{A}$ if $\mu(a)[p, q] = l$. A path π in $\mathcal{A}$ of length k is a sequence of transitions $t_1 t_2 \dots t_k$, where $t_i = (q_{i-1}, a_i, l_i, q_i)$. The word $a_1 a_2 \dots a_k$ is the *label* of π and it is denoted by $label(\pi)$. A path $\pi = t_1 t_2 \dots t_k$ is *accepting* if the first state of t_1 is an initial one, the last state of t_k is an accepting one and $l_i \neq 0_K$ for $i \in [k]$. An automaton $\mathcal{A}$ *accepts* a string w if there exists an accepting path $t_1 t_2 \dots t_k$ labeled by w. By $L(\mathcal{A})$ we denote the set of all strings accepted by $\mathcal{A}$ and we say that $\mathcal{A}$ *recognizes* the language $L(\mathcal{A})$. We call $\pi(i, j) = t_i t_{i+1} \dots t_{j-1}$ a *subpath* of π. Expression $\pi' \sqsubseteq \pi$ denotes that π' is a subpath of π. Subpath $\pi(i, j) = t_i \dots t_j$ is *non-empty* if $0 < i \leq j \leq k$ and it is a *a loop* if the source state of transition t_i and the destination state of t_j are the same.

An automaton $\mathcal{A}$ is *deterministic (or sequential)* if for each $a \in \Sigma, q \in Q$ there exists at most one $q \in Q$ such that $\mu(a)[p, q] \neq 0_K$ and the set of initial states I is a singleton. If the second condition is not satisfied, an automaton is called *finitely sequential* (fseq). Let $amb_{\mathcal{A}} : \Sigma^* \mapsto \mathbb{N}$ be the function which, for each string $w \in \Sigma^*$, assigns number of different accepting paths labeled by w to w. An automaton $\mathcal{A}$ is *finitely ambiguous* (famb) if there exists a nonnegative integer c such that $amb_{\mathcal{A}}(w) \leq c$ for all $w \in L(\mathcal{A})$. If $c \leq 1$ then finitely ambiguous automata are called *unambiguous* (namb). An automaton $\mathcal{A}$ is *polynomially ambiguous* (pamb) if there exists a polynomial p such that $amb_{\mathcal{A}}(w) < p(|w|)$. The classes of sequential automata and all automata are denoted by seq and rat.

In this paper we study only automata over the *tropical semiring* $\langle\{\mathbb{R} \cup \infty\} = \mathbb{R}_\infty, \min, +, \infty, 0\rangle$ earlier called as min-plus automata. The constructive part of this paper deals with the semiring with rational domain rather than real numbers, to deal with finite representations of numbers. A wfa $\mathcal{A}$ over the tropical semiring defines a function $S(\mathcal{A}) : \Sigma^* \to \mathbb{R}_\infty$ such that $S(\mathcal{A})(w) = \lambda \cdot \mu(w) \cdot \gamma$ for $w \in \Sigma^*$. Later, the value $S(\mathcal{A})(w)$ will be also denoted by $\langle w, \mathcal{A}\rangle$.

For the tropical semiring, the weight of a path is the sum of the weights of the transitions taken along the path, and the value of a word w is the minimal weight of an accepting path on it. Every nonaccepted string is mapped into ∞.

By $\mathsf{Seq}, \mathsf{Namb}, \mathsf{Fseq}, \mathsf{Famb}, \mathsf{Pamb}, \mathsf{Rat}$ we denote classes of functions described by automata which respectively belong to $\mathsf{seq}, \mathsf{namb}, \mathsf{fseq}, \mathsf{famb}, \mathsf{pamb}, \mathsf{rat}$.

In order to make the text more readable we use an arrow notation together with the notation of WFA introduced above. If we write $\xrightarrow{x} q$, it means $\lambda[q] = x$. A sequence $\xrightarrow{v_0} q_0 \xrightarrow{a_1|v_1} q_1 \xrightarrow{a_2|v_2} \cdots \xrightarrow{a_i|v_i} q_i$ denotes a path π being a sequence of transitions $t_1, \dots, t_i$ such that $label(\pi) = a_1 \dots a_i$, the transition t_j has weight v_j, and $\lambda[q_0] = v_0$.

A state $q \in Q$ is *accessible* if there exists string $w \in \Sigma^*$ such that $\lambda\mu(w)[q] \in \mathbb{R}$ and *coaccessible* if $\mu(w)\gamma[q] \in \mathbb{R}$. An automaton A is *trimmed* if all of its states are accessible and coaccessible. Two states p and q are said to be *siblings* if they satisfy $\rightarrow q_I \xrightarrow{u} p$ and $\rightarrow p_I \xrightarrow{u} q$ for some $u, v \in \Sigma^*$.

States $p, q \in Q$ are twins if for every words u_1, u_2 the following formula holds

$$\text{if } \xrightarrow{x_0} p_I \xrightarrow{u_1|x_1} p \xrightarrow{u_2|x_2} p \text{ and } \xrightarrow{y_0} q_I \xrightarrow{u_1|y_1} q \xrightarrow{u_2|y_2} q \text{ then } x_2 = y_2.$$

An automaton $\mathcal{A}$ has *twin property* if all of its pair of states p, q are twins. For a $t \in \Sigma^+$, a pair of states q_1, q_2 is $t-$*fork* if $q_1 \xrightarrow{t} q_1$ and $q_1 \xrightarrow{t} q_2$.

By the $[k]$ we denote the set $\{1, \dots, k\}$. An automaton $\mathcal{A} = \biguplus_{i=1}^{k} \mathcal{A}_i$ is a *disjoint sum* of automata $\mathcal{A}_i$ if

1. Q_i are pairwise disjoint sets of states and $Q = \bigcup_{i=1}^{k} Q_i$,
2. for each $i \in [k], q \in Q_i$ it holds that $\lambda_i(q) = \lambda(q)$ and $\gamma_i(q) = \gamma(q)$,
3. for each $i, j \in [k]$, $p \in Q_i$, $q \in Q_j$ if $i = j$ then $\mu(a)[p, q] = \mu_i(a)[p, q]$, else $\mu(a)[p, q] = \infty$.

3 Results

Given two words $u, v \in \Sigma^*$, by $lcp(u, v)$ we denote the longest common prefix of u and v. Define $d(u, v) = |u| + |v| - 2|lcp(u, v)|$. The automaton $\mathcal{A}$ is *N-Lipschitz* if

$$\forall u, v \in L(\mathcal{A}) \ \ |\langle u, \mathcal{A}\rangle - \langle v, \mathcal{A}\rangle| \leq N d(u, v)$$

and $\mathcal{A}$ is *Lipschitz* if it is N-Lipschitz for some $N \in \mathbb{N}^+$.

Lets consider an unambiguous trimmed automaton $\mathcal{A} = \langle Q, \Sigma, \lambda, \mu, \gamma\rangle$ over the tropical semiring.

Definition 1 [Fork Property]. *A trimmed unambiguous automaton $\mathcal{A}$ has the* Fork Property (FP) *if there exist states q_1, q_2 such that q_1 and q_2 are not twins and there exists a $t \in \Sigma^+$ such that q_1, q_2 is a $t-$fork.*

The proof of the next theorem is similar to the proof of non-emptiness of $\overline{\mathsf{Namb} \cap \mathsf{Fseq}}$ in subsection 3.3 of [8]. The new thing, compared to the mentioned proof is a generalization which states that any of the loops of the considered fork can have smaller weight and weights can be negative.

Theorem 2. *If $\mathcal{A}$ has the Fork Property then $S(\mathcal{A}) \notin \mathsf{Fseq}$.*

Proof. Assume that $\mathcal{A}$ has FP and q_1, q_2 is a pair of non twin states mentioned in FP. Therefore

$$\exists w \in \Sigma^* \exists q_I \in Q \qquad \xrightarrow{\epsilon|\lambda(q_I)} q_I \xrightarrow{w|c_1} q_1 \text{ and}$$

$$\exists v \in \Sigma^+ \exists x_1, x_2, x_1 \neq x_2 \quad q_1 \xrightarrow{v|x_1} q_1 \text{ and } q_2 \xrightarrow{v|x_2} q_2$$

and q_1, q_2 is a $t-$fork for some $t \in \Sigma^+$. For the sake of contradiction, assume that $S(\mathcal{A}) \in \mathsf{Fseq}$. There exist $\mathcal{B} \in \mathsf{fseq}$ such that $S(\mathcal{A}) = S(\mathcal{B})$ and $\mathcal{B} = \biguplus_{i=1}^{k} \mathcal{B}_i$, where $\mathcal{B}_i \in \mathsf{seq}$. Any $\mathcal{B} \in \mathsf{seq}$ is Lipschitz (see [10] for the proof of this fact). We will define a set of words such that any two words of this set cannot have its weight from the same $\mathcal{B}_i$, contradicting the fact that $\mathcal{B}_i$ is $N_{\mathcal{B}_i}-$ Lipschitz. Let z be any word that leads from q_2 to an accepting state q_F. Such a word exists because $\mathcal{A}$ is trimmed. Let c_3 be the weight of the path labeled by z. We define $\mathbf{W}$ as a set of words of the form $W_j := w\left(tv^{N_i}\right)_{i=1..j} z$, where $j \in [k+1]$. We describe later how to choose the constants N_i to suit our purpose. The weight $\langle W_j, \mathcal{A}\rangle$ equals

$$\lambda(q_I) + c_1 + (j-1)h_1 + h_2 + x_1\left(\sum_{i=1..j-1} N_i\right) + x_2 N_j + c_3 + \gamma(q_F),$$

where $h_1 = \mu(t)[q_1, q_1]$, $h_2 = \mu(t)[q_1, q_2]$. Due to the unambiguity of $\mathcal{A}$, all the W_j share the starting and ending state – if (p_s, W_i, p_e), (q_s, W_j, q_e) denote accepting paths labeled by W_i and W_j respectively starting at p_s, q_s and p_e, q_e then $p_s = q_s$ and $p_e = q_e$. Hence, for $j > l$

$$\begin{aligned}|\langle W_j, \mathcal{A}\rangle - \langle W_l, \mathcal{A}\rangle| &= \left| h_1(j-l) + x_1\left(\sum_{i=l\cdots j-1} N_i\right) + x_2 N_j - x_2 N_l\right| \\ &= \left| N_l(x_1 - x_2) + h_1(j-l) + x_1\left(\sum_{i=l+1\cdots j-1} N_i\right) + x_2 N_j\right| \\ &= \left|N_l(x_1 - x_2) + C(l,j)\right|,\end{aligned}$$

In the equation above, $C(l, j)$ depends on $N_{l+1}, \ldots, N_j$ and constants k, x_1, x_2, h_1.

We will now define the numbers N_i. Let $N = \max\{N_{\mathcal{B}_i} | i \in [k]\}$. Note that

$$d(W_j, W_l) = 2|z| + (j-l)|t| + |v| \sum_{i=l+1\cdots j} N_i$$

So $d(W_j, W_l)$ does not depend on N_l. Let us define the inequality for W_j, W_l as:

$$|\langle W_j, \mathcal{A}\rangle - \langle W_l, \mathcal{A}\rangle| > N d(W_j, W_l) \tag{1}$$

The numbers N_i will be defined in such way that inequality (1) holds for all pairs W_j, W_l, W_j and W_l preventing them from having it's weight from the same automaton B_i. Values of $N_1 \cdots N_{k+1}$ are defined inductively starting from $N_{k+1} = 1$. When $N_{m+1} \cdots N_{k+1}$ are already defined and all pairs W_j, W_l, for any $j > l$, $j, l \in \{m+1..k+1\}$ satify (1) we can define the next constant N_m.

The inequality for W_j and W_l, $j > l$ can be presented as:

$$|N_l(x_1 - x_2) + C(l,j)| > N \cdot \left(2|z| + (j-l)|t| + |v| \sum_{i=l+1\cdots j} N_i\right)$$

We will find $N_l \in \mathbb{R}^+$ so big, that

$$|N_l(x_1 - x_2) + C(r,j)| > \underbrace{N \cdot \left(2|z| + (k-l+1)|t| + |v| \sum_{i=l+1\cdots k+1} N_i\right)}_{=C_l}$$

for every r, j such that $r \geq l$ and $j \leq k+1$. Let $B = \min\{C(r,j) | r \geq l, j \leq k+1\}$. Then observe $N_l = \frac{C_l + |B| + 1}{|x_1 - x_2|}$ is sufficient to fulfill the inequalities. □

We have shown that the negation of the Fork Property is a necessary condition for having a finitely sequential equivalent if the given automaton is unambiguous. Now we will show that the negation of the Fork Property is also sufficient.

Definition 3 [Critical Pair]. *Let*

$$D_{\mathcal{A}} := \{\langle q_1, q_2\rangle \in Q \times Q \mid q_1 \textit{ and } q_2 \textit{ are siblings, not twins in } \mathcal{A}\}.$$

A pair $\langle q, E\rangle \in Q \times P(Q)$ is said to be critical if $\exists p \in Q$ $\langle q, p\rangle \in D_{\mathcal{A}}$ and $\{q, p\} \subseteq E$.

We define $\hat{\mathcal{A}}$ as $\langle \hat{Q}, \Sigma, \hat{\lambda}, \hat{\mu}, \hat{\gamma}\rangle$, where:

$$\hat{Q} := Q \times P(Q)$$

$$\hat{\lambda}(\langle q, E\rangle) := \begin{cases} \lambda(q), & \text{if } E = \{r \in Q \mid \lambda(r) \neq \infty\} \\ \infty & \text{otherwise} \end{cases}$$

$$\hat{\gamma}(\langle q, E\rangle) := \gamma(q)$$

$$\hat{\mu}(a)\left(\langle q_1, E_1\rangle, \langle q_2, E_2\rangle\right) := \begin{cases} \mu(a)(q_1, q_2), & \text{if } q_1 \in E_1, \text{ and } \langle q_1, E_1\rangle \text{ is not} \\ & \text{critical and} \\ & E_2 = \{q \mid \exists p \in E_1 \quad \mu(a)(p,q) \neq \infty\} \\ \mu(a)(q_1, q_2), & \text{if } q_1 \in E_1 \quad \wedge \langle q_1, E_1\rangle \text{ is critical} \\ & \text{and } E_2 = \{q \mid \mu(a)(q_1, q) \neq \infty\} \\ \infty & \text{otherwise} \end{cases}$$

The intuition behind above definitions is the following: The first component simulates the automaton $\mathcal{A}$. The automaton starts from the initial states which holds all initial states in the second component. The fact that the current state is critical means that a non-twin pair of states has been detected and one of the states is our current first component. Following the transitions we accumulate in the second component all states that can be reached in $\mathcal{A}$ from the last critical state. When we are in the new critical state we reduce the second component to the single state remembering only this first component which currently is the cause of non-twin pair occurrence.

Note that if $q_0 \xrightarrow{a_0|t_0} q_1 \xrightarrow{a_1|t_1} \cdots \xrightarrow{a_{i-1}|t_{i-1}} q_i$ is a path in $\mathcal{A}$ then for some sequence $(E_j)_{j=1,..,i}$, $\langle q_0, E_0\rangle \xrightarrow{a_0|t_0} \langle q_1, E_1\rangle \xrightarrow{a_1|t_1} \cdots \xrightarrow{a_{i-1}|t_{i-1}} \langle q_i, E_i\rangle$ is a path in $trim(\hat{\mathcal{A}})$. In addition, when we have a path in the automaton $trim(\hat{\mathcal{A}})$, by mapping all its states with $\pi_f(\langle q, E\rangle) := q$ we obtain a valid path in $\mathcal{A}$. Hence we obtain the following.

Remark 4. $S(\mathcal{A}) = S(trim(\hat{\mathcal{A}}))$.

The function $\hat{\mu}$ is defined such that for each $\langle q_1, E_1\rangle$ if $\mu(a)(q_1, q_2) \neq \infty$ then the E_2 such that $\hat{\mu}(a)(\langle q_1, E_1\rangle, \langle q_2, E_2\rangle) \neq \infty$ is uniquely determined. Hence nonambiguity is preserved for $\hat{\mathcal{A}}$.

Remark 5. $trim(\hat{\mathcal{A}}) \in \mathsf{namb}$.

Let us now show a crucial property of $trim(\hat{\mathcal{A}})$.

Lemma 6. *An automaton $\mathcal{A}$ has the Fork Property iff for all $m \in \mathbb{N}$ there exists an accepting path in $trim(\hat{\mathcal{A}})$ with more than m critical states.*

Proof. Assume that $\mathcal{A}$ has FP. Moreover, assume that q_1, q_2 are the states from the property definition. Let w be the word leading to the state q_1 and v be a word leading from q_2 to a final state. Consider the word $s_m := wt^{m+2}v$. The word s_m is accepted by $\mathcal{A}$ on the path $\pi : q_0 \xrightarrow{w} q_1 \left(\xrightarrow{t} q_1\right)^m \xrightarrow{t} q_2 \xrightarrow{v} q_f$. The path π has a corresponding path $\overline{\pi}$ in $trim(\hat{\mathcal{A}})$, where

$$\overline{\pi} = \langle q_0, E_0\rangle \xrightarrow{w} \langle q_1, E_1\rangle \left(\xrightarrow{t} \langle q_1, E_{2\cdots m+2}\rangle\right)^{m+1} \xrightarrow{t} \langle q_2, E_{m+3}\rangle \xrightarrow{v} \langle q_f, E_f\rangle.$$

Let us consider the subpath of $\overline{\pi}$ starting after $\langle q_1, E_i\rangle$ and ending with $\langle q_1, E_{i+1}\rangle$. If all the states on this subpath before $\langle q_1, E_{i+1}\rangle$ are non-critical, then, by the construction of $trim(\hat{\mathcal{A}})$, $\{q_1, q_2\} \subseteq E_{i+1}$ and so $\langle q_1, E_{i+1}\rangle$ is critical. Therefore there is at least one critical state on the subpath. The path $\overline{\pi}$ contains $m+1$ such subpaths with at least one critical state on each one, so it has more than m critical states. It is also an accepting path. This proves the first implication.

Assume that for every $m \in \mathbb{N}$ there exists an accepting path in $trim(\hat{\mathcal{A}})$ with more than m critical states. Let us take such path for $m = |Q| + 1$. It contains at least one pair of critical states $\langle q_0, E_0\rangle$ and $\langle q_1, E_1\rangle$ with the same first coordinate $q_0 = q_1$. As $\langle q_1, E_1\rangle$ is critical, there exist $p_1 \in E_1$ such that q_1 and p_1 are siblings and are not twins. By the definition of $\hat{\mathcal{A}}$ if $\langle q_0, E_0\rangle$ is critical then all states in E_1 are reachable from q_0 in $\mathcal{A}$ by some word w. Thus, there is a path from $\langle q_0, E_0\rangle$ to $\langle p_1, H_1\rangle$ for some H_1 labeled with the same word as our path from $\langle q_0, E_0\rangle$ to $\langle q_1, E_1\rangle$. Therefore in $\mathcal{A}$, $q_0 \xrightarrow{w} q_1$ and $q_0 \xrightarrow{w} p_1$, and so $\mathcal{A}$ has the Fork Property. □

The Lemma 6 says that if the Fork Property does not hold, then only short sequences of critical states can appear on a single path of $trim(\hat{\mathcal{A}})$. The number of critical states is bounded by $m = |Q| + 1$. This gives the intuition that all accepting paths can be split between components which contain paths with the same sequence of critical states. Since the length of the sequence is bounded, only a finite number of components must be considered. The main purpose of such split between components is to obtain components satisfying the Twin Property, which can then be determinized. This intuition is correct but insufficient to ensure Twin Property of the component because several paths with the same label may contain the same maximal subsequence of critical states. Hence the sequence of critical states will be strengthened by adding to each critical state a number to distinguish between different paths labeled by the same string and the same sequence of critical states.

The rest of this section is devoted to the formal construction of the split described above and the proof that each component we construct has the Twin Property. In the proof below we are focused on correctness of the formal construction. Correctness means that each component is unambiguous and has the Twin Property. Completeness, meaning that the outcome represents the same function, is straightforward.

It would suffice to take $m = |Q| + 1$ in the proof of Lemma 6. Let k be a positive integer not greater than m. Let $R \subseteq Q \times \mathcal{P}(Q) \times [m]$ be the maximal set such that if $\langle p, E, j\rangle \in R$ then $\langle p, E\rangle$ is critical. An $\boldsymbol{a} \in R^k$ is a sequence of critical pairs of $trim(\hat{\mathcal{A}})$ *marked* by numbers from $[m]$. By $\boldsymbol{a}[i]$ we denote the i'th element of the sequence. If $\boldsymbol{a}[i] = \langle p, E, j\rangle$ by $\boldsymbol{a}^1[i], \boldsymbol{a}^2[i], \boldsymbol{a}^3[i]$ we denote respectively p, E, j.

If $q = \langle \boldsymbol{c}^1[i], \boldsymbol{c}^2[i]\rangle$ and $\boldsymbol{c}^3[j] = 0 \iff j \geq i$ we call i the *q-upgrade* of $\boldsymbol{c}$ we also say that such a q *extends* $\boldsymbol{c}$. A vector $\boldsymbol{b} \in R^k$ is *decent* if it satisfies $\forall i \in [k] \ \left(\boldsymbol{b}^3[i] = 0 \Rightarrow \forall j > i \ \boldsymbol{b}^3[j] = 0\right)$. A decent vector $\boldsymbol{c}$ *subsumes* a decent vector $\boldsymbol{b}$, what is denoted also by $\boldsymbol{b} \preceq \boldsymbol{c}$ if $\forall i \in [k] \ \boldsymbol{b}^3[i] \neq 0 \Rightarrow \boldsymbol{b}^3[i] = \boldsymbol{c}^3[i]$. A vector $\boldsymbol{c} \in R^k$ is an ***a**-type*, if $\langle \boldsymbol{c}^1[i], \boldsymbol{c}^2[i]\rangle = \langle \boldsymbol{a}^1[i], \boldsymbol{a}^2[i]\rangle$ for all $i \in [k]$, and it is decent. A vector $\boldsymbol{c} \in R^k$ is an ***a**-verge* if it is $\boldsymbol{a}-$type and $\boldsymbol{c} \preceq \boldsymbol{a}$.

Fix $\boldsymbol{a} \in R^k$ such that $0 < \boldsymbol{a}^3[i] \leq m$ for all $i \in [k]$. By $\mathcal{A}[\boldsymbol{a}] = \langle \Sigma, Q^{\boldsymbol{a}}, \lambda^{\boldsymbol{a}}, \mu^{\boldsymbol{a}}, \gamma^{\boldsymbol{a}}\rangle$ we denote an $\boldsymbol{a}-$*split* automaton defined as follows: $Q^{\boldsymbol{a}} = \hat{Q} \times R^k \times \mathcal{P}(\hat{Q} \times R^k)$.

Definition 7 [Context Extension]. *Let $\boldsymbol{c} \in R^k$ be an $\boldsymbol{a}-$type vector and let $A \in \mathcal{P}(\hat{Q} \times R^k)$. Let $q \in \hat{Q}$ be a non-critical state or a critical state which extends $\boldsymbol{c}$. An $\boldsymbol{a}-$type vector $\boldsymbol{b}$ is an* A-context extension of $\boldsymbol{c}$ *by q, denoted by $\boldsymbol{b} = \boldsymbol{c} + q[A] \in R^k$ if*

1. *$\boldsymbol{b} = \boldsymbol{c}$ and q is non-critical or*
2. *q is the $i-$upgrade of $\boldsymbol{c}$ and for all $l \in [k]$*

$$\boldsymbol{b}^3[l] = \begin{cases} \min([m] \setminus \{j > 0 \mid \exists \langle p, \boldsymbol{d}\rangle \in A \ \boldsymbol{d}^3[i] = j\}) & l = i, \\ \boldsymbol{c}^3[l] & \textit{otherwise.} \end{cases}$$

We would like to emphasize here that the expression $\boldsymbol{b} = \boldsymbol{c} + q[A]$ is a relation between $\boldsymbol{b}, \boldsymbol{c}, q$, and A. In particular if q is a critical state which do not extends $\boldsymbol{c}$, $\neg(\boldsymbol{b} = \boldsymbol{c} + q[A])$.

Let $T^{\boldsymbol{a}}$ be the set of all $\boldsymbol{a}-$type elements in R^k.

Definition 8 [Context Successor]. *For $b \in \Sigma$, let $ns_b : \mathcal{P}(\hat{Q} \times T^{\boldsymbol{a}}) \to \mathcal{P}(\hat{Q} \times T^{\boldsymbol{a}})$ be a function such that for given A, $ns_b(A) = B$ if*

$$B = \{\langle q, \boldsymbol{b}\rangle \in \hat{Q} \times T^{\boldsymbol{a}} \mid \exists \langle p, \boldsymbol{c}\rangle \in A \;\; \hat{\mu}(b)(p,q) \neq \infty \wedge \boldsymbol{b} = \boldsymbol{c} + q[A]\}.$$

*Set B is called $b-$*successor of context *A, and ns_b is called $b-$*successor function.

Note that such definition of ns_b ensures that the number marking a vector is not greater than m. We have to emphasize also that the definition of B is correct as well as the next definition because we are restricted to the unambiqous automaton $\mathcal{A}$ and moreover there is at most one $\langle p, \boldsymbol{b}\rangle \in A$ which satisfies $\hat{\mu}(a)(p,q) \neq \infty \wedge \boldsymbol{b} = \boldsymbol{a} + q[A]$.

Now we are ready to define functions μ, λ, γ for $\mathcal{A}[\boldsymbol{a}]$. For $Q = \langle q, \boldsymbol{c}, A\rangle$ such that $\boldsymbol{c}$ is an $\boldsymbol{a}$-type

$$\lambda^{\boldsymbol{a}}(Q) := \begin{cases} \hat{\lambda}(q), & \text{if } \sum_{i=1}^{k} \boldsymbol{c}^3[i] = 0, \\ & A = \{\langle p, \boldsymbol{b}\rangle : \sum_{i=1}^{k} \boldsymbol{b}^3[i] = 0, \hat{\lambda}(p) \neq \infty, \boldsymbol{b} \text{ is an } \boldsymbol{a} - \text{type}\} \\ \infty & \text{otherwise} \end{cases}$$

$$\gamma^{\boldsymbol{a}}(\langle q, \boldsymbol{a}, A\rangle) := \hat{\gamma}(q)$$

For $Q_1 = \langle q_1, \boldsymbol{a}_1, A_1\rangle, Q_2 = \langle q_2, \boldsymbol{a}_2, A_2\rangle$ we define

$$\mu^{\boldsymbol{a}}(b)\,(Q_1, Q_2) := \begin{cases} \hat{\mu}(b)(q_1, q_2), & \text{if } A_2 = ns_b(A_1); \boldsymbol{a}_2 = \boldsymbol{a}_1 + q_2[A_1]; a_1, a_2 \preceq \boldsymbol{a} \\ \infty & \text{otherwise} \end{cases}$$

In the special case of an $\mathcal{A}[\boldsymbol{a}]$ automaton, when $k = 0$ and $\boldsymbol{a}$ has no co-ordinates, we write $\boldsymbol{a} = \boldsymbol{\epsilon}$. In this case all nonzero transitions are between non-critical states. In our construction of automata $\mathcal{A}[\boldsymbol{a}]$ all paths of $trim(\hat{\mathcal{A}})$ are distributed between $\mathcal{A}[\boldsymbol{a}]$ without changing weights of transitions. Therefore each path in $trim(\hat{\mathcal{A}})$ has his counterpart in $\mathcal{C} = \biguplus_{k \in [m]} \biguplus_{\boldsymbol{a} \in R^k \cup \{\boldsymbol{\epsilon}\}} \mathcal{A}[\boldsymbol{a}]$ where consecutive states in the first path are equal to the first coordinate of the corresponding states in the second one. The fact that each of the paths in $\hat{\mathcal{A}}$ belongs to some $\mathcal{A}[\boldsymbol{c}]$ depends on the sequence of critical states occurring on the path and the marking numbers given by the context extension operation. Since for each word w number of paths labeled by w is at most m a particular path labeled by w gets its marking numbers for the critical states. Hence each of paths are enclosed to some $\mathcal{A}[\boldsymbol{c}]$.

Remark 9. *$S(trim(\hat{\mathcal{A}})) = S(\mathcal{C})$ and $\forall \boldsymbol{a} \in \bigcup_{k \in [m]} R^k \cup \{\boldsymbol{\epsilon}\}$, $\mathcal{A}[\boldsymbol{a}] \in$ namb.*

From the fact that definition of $\mu^{\boldsymbol{a}}$ allows only vectors which are subsumed by $\boldsymbol{a}$, we also get the following.

Fact 10. *For every $w \in \Sigma^+$ and every pair of different paths if $\xrightarrow{x_0} p_0 \xrightarrow{w|x_1} \langle p, \boldsymbol{b}, A\rangle$ and $\xrightarrow{y_0} q_0 \xrightarrow{w|y_1} \langle q, \boldsymbol{c}, B\rangle$ in $\mathcal{A}[\boldsymbol{a}]$ then $\boldsymbol{c} \preceq \boldsymbol{b}$ or $\boldsymbol{b} \preceq \boldsymbol{c}$.*

Now, we can prove our main technical result.

Theorem 11. *If $\mathcal{A} \in$ namb does not satisfy the Fork Property, then $S(\mathcal{A}) \in$ Fseq.*

Proof. By the previous remarks we have that

$$S(\mathcal{A}) = \dot{\bigcup_{k\in[m]}} \dot{\bigcup_{\boldsymbol{a}\in R^k\cup\{\boldsymbol{\epsilon}\}}} \mathcal{A}[\boldsymbol{a}],$$

and all of $\mathcal{A}[\boldsymbol{a}]$ are namb. Now we need to show that any $\mathcal{A}[\boldsymbol{a}]$ has twin property and therefore all of them are determinizable. This proves $S(\mathcal{A}) \in$ Fseq.

For the sake of contradiction assume that $\mathcal{A}[\boldsymbol{a}]$ has no twin property. That means that there are $p, q \in Q^{\boldsymbol{a}}$ and $w, v \in \Sigma^*$ such that

$$\infty \neq x_2 = \mu^{\boldsymbol{a}}(v)(q,q) \neq \mu^{\boldsymbol{a}}(v)(p,p) = y_2 \neq \infty,$$

and for some $q_0, p_0 \in Q^{\boldsymbol{a}}$ there are two different paths

$$\pi_1 \xRightarrow{x_0} q_0 \xrightarrow{w|x_1} p \xrightarrow{v|x_2} p \text{ and } \pi_2 \xRightarrow{y_0} p_0 \xrightarrow{w|y_1} q \xrightarrow{v|y_2} q.$$

Assume that π_1 and π_2 can be presented as $\pi_1 = \alpha_1\alpha_2\alpha_3\alpha_4, \pi_2 = \alpha_1\beta_2\beta_3\beta_4$, where

1. $|\beta_i| = |\alpha_i|$ for $i = 2, 3, 4$ and $|\alpha_4| = |v|$;
2. $\langle p', \boldsymbol{c}, A\rangle$, and $\langle q', \boldsymbol{d}, B\rangle$ are respectively the first states of α_3 and β_3 and at the same time the earliest mutual states such that $\boldsymbol{c} \neq \boldsymbol{d}$;
3. $\langle p'', \boldsymbol{e}, A\rangle$, and $\langle q'', \boldsymbol{e}, B\rangle$ are respectively the first states of α_2 and β_2 and at the same time the earliest mutual states such that $p'' \neq q''$.

First note that states $\langle p', \boldsymbol{c}, A\rangle$, and $\langle q', \boldsymbol{d}, B\rangle$ have to appear on the paths not later than states p, q and $\langle p'', \boldsymbol{e}, A\rangle$, $\langle q'', \boldsymbol{e}, B\rangle$ appears not later than $\langle p', \boldsymbol{c}, A\rangle$, $\langle q', \boldsymbol{d}, B\rangle$. Now remark that the second components of all states, excluding the last one, on the paths α_2 and β_2 are equal $\boldsymbol{e}$. Otherwise $\mathcal{A}[\boldsymbol{a}]$ would not be unambiguous, as each change to the other vector $\boldsymbol{f}$, according to the assumptions, can be done simultaneously for both paths extending e by the same state r. But this means that there are two different paths leading to the state r in $\hat{\mathcal{A}}$. Therefore there are no states in α_2 and β_2 with critical first component.

By Fact 10 either $\boldsymbol{c} = \boldsymbol{e}$ or $\boldsymbol{d} = \boldsymbol{e}$. Without loss of generality assume that $\boldsymbol{c} = \boldsymbol{e}$. By the definition of $\mu^{\boldsymbol{a}}$, in particular due to the definition of the operation of context extension ($\boldsymbol{a_2} = \boldsymbol{a_1} + q[A]$), all states in $\alpha_3\alpha_4$ cannot be extended being still subsumed by $\boldsymbol{a}$. This is because that if i is an upgrade of $\boldsymbol{c}$ then it cannot be an upgrade of any $\boldsymbol{f}$ such that $\boldsymbol{d} \preceq \boldsymbol{f}$, just context extension assigns a number to $\boldsymbol{c}^3[i]$ different then upgraded earlier $\boldsymbol{f}^3[i]$. Now we have a complete picture of which paths in $\hat{A}$ are covered by $\mathcal{A}[\boldsymbol{a}]$ – only that which are $\boldsymbol{a}-$verge.

This means that all first components of states in $\alpha_3\alpha_4$ are non-critical. Note that neither p'' nor q'' are critical states, otherwise because, they are different, we would have contradiction with the second item of the assumption. Let $p = \langle\langle s_1, E_1\rangle, \boldsymbol{e}, A\rangle$ and $q = \langle\langle s_2, E_2\rangle, \boldsymbol{e}', B\rangle$. Now, by the definition of $\hat{\mathcal{A}}$, $s_2 \in E_1$ but $s_1 \xrightarrow{v|x_2} s_1$ and $s_2 \xrightarrow{v|y_2} s_2$ are the subpaths in $\mathcal{A}$. This contradicts the previous conclusion that $\langle s_1, E_1\rangle$ is not critical in $\hat{\mathcal{A}}$. □

Complexity. It is decidable in polynomial time for unambiguous automata if given p, q are twins [1]. Therefore one can iterate the decision procedure for all pairs and in each step the reachability condition which appears in the fork property can be tested. Hence the Fork Property and its negation can be decided in polynomial time.

The naive estimation of the upper bound for the size of one component in $\mathcal{C}$ can get $m^3 2^{O(m^3)}$. This is exponential in the size of the input because $m = |Q|+1$. The number of components in $\mathcal{C}$ can be bounded by $m^2 2^{O(m^2)}$. Therefore deciding if an unambiguous automaton is finitely sequential can be done in PTIME and the counterpart is at most doubly exponential in the size of the input - because components of $\mathcal{C}$ need to be determinized.

Acknowledgement. We thank Lukasz Kaiser for bunch of remarks concerning the text improvement. We are grateful to Daniel Kirsten for sharing the 'almost full version' of the paper [7] and for discussions during the first author's visit at Humboldt University and Liafa Laboratory.

References

1. Allauzen, C., Mohri, M.: Efficient algorithms for testing the twins property. Journal of Automata, Languages and Combinatorics 8(2), 117–144 (2003)
2. Allauzen, C., Riley, M.D., 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)
3. Aminof, B., Kupferman, O., Lampert, R.: Rigorous approximated determinization of weighted automata. In: LICS. pp. 345–354. IEEE Computer Society (2011)
4. Choffrut, C.: Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles. Theor. Comput. Sci. 5(3), 325–337 (1977)
5. Kirsten, D.: A burnside approach to the termination of mohri's algorithm for polynomially ambiguous min-plus-automata. ITA 42(3), 553–581 (2008)
6. Kirsten, D.: Decidability, undecidability, and pspace-completeness of the twins property in the tropical semiring. Theor. Comput. Sci. 420, 56–63 (2012)
7. Kirsten, D., Lombardy, S.: Deciding unambiguity and sequentiality of polynomially ambiguous min-plus automata. In: STACS. LIPIcs, vol. 3, pp. 589–600. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)
8. Klimann, I., Lombardy, S., Mairesse, J., Prieur, C.: Deciding unambiguity and sequentiality from a finitely ambiguous max-plus automaton. Theor. Comput. Sci. 327(3), 349–373 (2004)
9. Krob, D., Litp, P.: Some consequences of a fatou property of the tropical semiring. J. Pure Appl. Algebra 93, 231–249 (1994)
10. Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23(2), 269–311 (1997)
11. Mohri, M., Pereira, F., Riley, M.: Weighted finite-state transducers in speech recognition. Computer Speech & Language 16(1), 69–88 (2002)
12. Sakarovitch, J., de Souza, R.: On the decomposition of k-valued rational relations. In: STACS. LIPIcs, vol. 1, pp. 621–632. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2008)

Duplication-Loss Genome Alignment: Complexity and Algorithm

Billel Benzaid[2], Riccardo Dondi[1], and Nadia El-Mabrouk[2]

[1] Dipartimento di Scienze Umane e Sociali, Università degli Studi di Bergamo
Via Donizetti 3, 24129 Bergamo - Italy
[2] Départment d'Informatique et Recherche Opérationnelle, Université de Montréal
Pavillon André-Aisenstadt, CP 6128 succ Centre-Ville, Montréal, Québec, Canada
riccardo.dondi@unibg.it, {benzaidb,mabrouk}@iro.umontreal.ca

Abstract. Recently, an Alignment approach for the comparison of two genomes, based on an evolutionary model restricted to Duplications and Losses, has been presented. An exact linear programming algorithm has been developed and successfully applied to the Transfer RNA (tRNA) repertoire in Bacteria, leading to interesting observation on tRNA shift of identity. Here, we explore a direct dynamic programming approach for the Duplication-Loss Alignment of two genomes, which proceeds in two steps: (1) (The Dynamic Programming step) Outputs a best candidate alignment between the two genomes and (2) (Minimum Label Alignment problem) Finds an evolutionary scenario of minimum duplication-loss cost that is in agreement with the alignment. We show that the Minimum Label Alignment is APX-hard, even if the number of occurrences of a gene inside a genome is bounded by 5. We then develop a heuristic which is a thousands of times faster than the linear programming algorithm and exhibits a high degree of accuracy on simulated datasets. The heuristic has been implemented in JAVA and is available on request.

Keywords: Comparative Genomics, Genome Alignment, Duplication, Algorithms, Computational Complexity, Dynamic Programming.

1 Introduction

The abundance of completely sequenced and annotated genomes present in public repositories has reinforced the role of genome comparison as the primary approach to gain insight in the evolution of genomes and gene families. When comparing complete genomes, the mutations of interest are macro-evolutionary events such as rearrangements (inversions, transpositions, translocations etc.) and content modifying operations (duplications, losses, horizontal gene transfer etc.) affecting the overall organization of genes, rather than micro-evolutionary events, such as single nucleotide substitutions, affecting single gene sequences. In other words, genomes are modeled as strings of characters over an alphabet Σ of gene families. The case of strings being permutations (i.e. each gene family with a single representative in each genome) has been largely considered by the genome rearrangement community for pairwise comparison (for example [3,8,10,13]) or multiple comparison in a phylogenetic framework (for example [4,12,14,15]). An extra degree of difficulty is introduced in the case of strings containing multiple gene copies. Most of the methods used for comparing two genomes with duplicates (reviewed in [6,7,9]) rely mainly on rearrangement events. Contrariwise, we considered in [11], an evolutionary model restricted to content-modifying operations, and more specifically to duplications and losses. We showed that this model is required to study the evolution of certain gene families, such as Transfer RNAs (tRNAs). From a combinatorial point of view, the main consequence of ignoring rearrangements is the fact that gene organization is preserved, which allows reformulating the comparison of two genomes as a Duplication-Loss Alignment problem: find an alignment minimizing the cost of duplications and losses. As in [11], we consider in this paper the cost of an alignment to be the number of underlying segmental duplications (duplication of a string of adjacent genes) and single losses (loss of a single gene). Although alignments are *a priori* simpler to handle than rearrangements, we showed in [11] that a direct approach based on dynamic programming leads, at best, to an efficient heuristic, and we rather developed an exact pseudo-boolean linear programming algorithm. This algorithm is however exponential in the worst case, preventing from being applicable to relatively large genomes (more than 200 genes). The problem has actually been recently shown to be NP-hard [5].

In this paper, we further explore the suggested direct dynamic programming approach, which is in two steps: (1) (The Dynamic Programming step) Output a best candidate alignment between the two genomes and (2) (Minimum Label Alignment problem) Find an evolutionary scenario of minimum duplication-loss cost that is in agreement with the alignment. The way to solve the Minimum Label Alignment problem, as well as its complexity, were left open in [11]. We show in Section 4 that it is APX-hard, even if the number of occurrences of a gene inside a genome is bounded by 5. We then, in Section 5, present a heuristic for the Duplication-Loss Alignment problem, which is a thousands of times faster than the linear programming algorithm and exhibits optimal or near-optimal results on simulated datasets obtained with an evolutionary model consistent with that observed for the tRNA repertoire in *Bacillus*.

We begin by introducing the notations and alignment problems in Section 2, and the dynamic programming approach in Section 3.

2 Preliminaries

Strings: We consider single chromosomal (circular or linear) genomes, represented as gene orders with duplicates. More precisely, given an alphabet Σ, each character representing a specific gene family, a *genome* or *string* is a sequence of characters from Σ, where each character may appear many times. As the content-modifying operations considered in this paper do not change gene orientation, we can assume w.l.o.g. that genes are unsigned. For example, X in Figure 1 is a genome on the alphabet $\Sigma = \{a, b, c, d, e, f\}$, with four gene copies from the gene family identified by b, and a single copy from family f.

Given a string Z, we denote by $|Z|$ its length, by $Z[i]$, $1 \leq i \leq |Z|$, the i-th character of Z, and by $Z[i,j]$, $1 \leq i \leq j \leq |Z|$, the substring of Z that starts at position i and ends at position j. Finally, two substrings $Z[i_1, i_2]$ and $Z[j_1, j_2]$, $1 \leq i_2 \leq j_2 \leq |Z|$, overlap if $j_1 \leq i_2$.

The Duplication-Loss Model of Evolution: We assume that present-day genomes have evolved from an ancestral string through duplications and losses, where: (i) A *Duplication* of size k is an operation that copies a substring of size k of a current genome X somewhere else in the genome. Given two identical non overlapping substrings $X[i, i+k-1]$ and $X[j, j+k-1]$ of X, we denote by $D = (X[i, i+k-1], X[j, j+k-1])$ a *duplication* from $X[i, i+k-1]$ to $X[j, j+k-1]$; the string $X[i, i+k-1]$ is called the *source*, and the string $X[j, j+k-1]$ the *target* of the duplication D; (ii) A *loss* of size k is an operation $L = (X[i, i+k-1])$ that removes a substring $X[i, i+k-1]$ of size k from genome X.

Consider a duplication $D = (X[i_1, i_2], X[j_1, j_2])$ of X. Such a duplication is called *maximal* if it cannot be extended using positions adjacent to the source and target substrings.

Given an integer $k \geq 1$, the cost of a duplication of size k is denoted by $c(D(k))$, and the cost of a loss of size k is denoted by $c(L(k))$.

The Duplication-Loss Alignment Problem: We introduced in [11] the concept of "Feasible" Labeled Alignment of two genomes X and Y, and showed the one-to-one correspondence between the set of such alignments and the set of all possible "visible" evolutionary histories from a common ancestor A to X and Y. Definitions on alignments are given below, and illustrated in Figure 1.

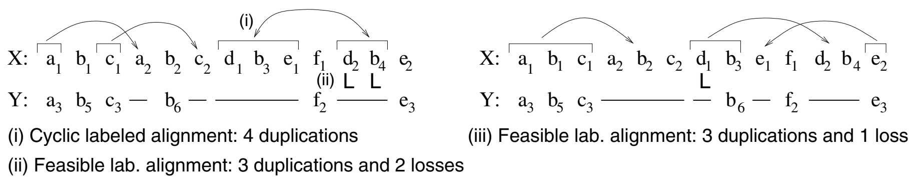

(i)
X: a1 b1 c1 a2 b2 c2 d1 b3 e1 f1 d2 b4 e2
(ii) L L
Y: a3 b5 c3 — b6 — ——— f2 ——— e3
(i) Cyclic labeled alignment: 4 duplications
(ii) Feasible lab. alignment: 3 duplications and 2 losses
X: a1 b1 c1 a2 b2 c2 d1 b3 e1 f1 d2 b4 e2
L
Y: a3 b5 c3 ——— — b6 — f2 ——— e3
(iii) Feasible lab. alignment: 3 duplications and 1 loss

Fig. 1. Labeled alignments for strings X = "abcabcdbefdbe" and Y = "abcbfe". Costs are $c(D(k)) = 1$ and $c(L(k)) = k$ for any integer k. Losses are denoted by "L" and duplications by arrows from source (indicated by bracket) to target. Two different labeling are given for the left alignment: one (i) with "$d_2\, b_4$" being interpreted as the target of a duplication, and one (ii) with the same substring interpreted as two losses.

In the remaining of this paper, we consider two genomes X and Y on an alphabet Σ, with $|X| = n$ and $|Y| = m$. Let $\Sigma^- = \Sigma \cup \{-\}$ be the alphabet Σ augmented with an additional character '-' called a gap.

Definition 1. *An* Alignment *of X and Y is a pair $(\mathcal{X}, \mathcal{Y})$ of strings on $\Sigma^- \times \Sigma^-$ obtained by filling X and Y respectively with gaps, such that the resulting* Aligned

Genomes $\mathcal{X}$ *and* $\mathcal{Y}$ *are equal length. Moreover, each position* i*, with* $1 \leq i \leq |\mathcal{X}|$*, is such that either* $\mathcal{X}[i] = \mathcal{Y}[i] \neq -$ *(position* i *is called a* Match*), or exactly one of* $\mathcal{X}[i]$*,* $\mathcal{Y}[i]$ *is equal to a gap (position* i *is called a* Mismatch*).*

In order to uniquely match an alignment $(\mathcal{X}, \mathcal{Y})$ with a duplication-loss history leading to X and Y from a common ancestor, we need to label unmatched characters of the aligned genomes $\mathcal{X}$ and $\mathcal{Y}$ in terms of *duplications* and *losses*.

Definition 2. *A* Labeling $\mathcal{L}(\mathcal{X})$ *of an aligned genome* $\mathcal{X}$ *(or simply* $\mathcal{L}$ *if no ambiguity) is a set of losses and duplications, such that for each mismatched position* j*,* $1 \leq j \leq |\mathcal{X}|$*,* $\mathcal{L}(\mathcal{X})$ *contains either a loss* $L = (\mathcal{X}[j_1, j_2])$ *or exactly one duplication* $D = (\mathcal{X}[i_1, i_2], \mathcal{X}[j_1, j_2])$*, with* $1 \leq j_1 \leq j \leq j_2 \leq |\mathcal{X}|$*.*

Now, a Labeling *of an alignment* $(\mathcal{X}, \mathcal{Y})$ *is a pair* $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ *where* $\mathcal{L}(\mathcal{X})$ *and* $\mathcal{L}(\mathcal{Y})$ *are labeling of* $\mathcal{X}$ *and* $\mathcal{Y}$ *respectively. The pair* $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ *is a* Labeled Alignment *of* X *and* Y*. The cost of a labeling* $\mathcal{L}(\mathcal{X})$ *is the cost of the underlying operations (losses and duplications). The cost of a labeled alignment* $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ *is the sum of cost of the two labeling* $\mathcal{L}(\mathcal{X})$ *and* $\mathcal{L}(\mathcal{Y})$*.*

The above definition is not sufficient to ensure a correct interpretation of an alignment in term of duplication-loss history, as it does not prevent from a "cyclic" interpretation of an alignment. For example the labeled alignment (i) in Figure 1 is not feasible as it reflects a history with two circular duplications $D = (d_1b_3e_1, d_2b_4e_2)$ and $D' = (d_2b_4, d_1b_3)$. A "feasible labeling" is a non-cyclic labeling, where cycles are rigorously defined as follows.

Definition 3. *Consider a set of duplications* $\mathcal{D}$*.* $\mathcal{D}$ *induces a* Duplication Cycle *if there is a permutations* $D_1 = (\mathcal{X}[i_1, r_1], \mathcal{X}[j_1, s_1])$*,* $D_2 = (\mathcal{X}[i_2, r_2], \mathcal{X}[j_2, s_2])$*,* $\ldots$*,* $D_h = (\mathcal{X}[i_h, r_h], \mathcal{X}[j_h, s_h])$ *of the duplications in* $\mathcal{D}$*, such that the substrings* $\mathcal{X}[j_p, s_p]$ *and* $\mathcal{X}[i_{p+1}, r_{p+1}]$ *overlap, for each* $1 \leq p \leq h - 1$*, and the substrings* $\mathcal{X}[j_h, s_h]$ *and* $\mathcal{X}[i_1, r_1]$ *overlap.*

Now, a labeling $\mathcal{L}(\mathcal{X})$ is Feasible if there is no subset of duplications in $\mathcal{L}(\mathcal{X})$ that induces a duplication cycle. Finally a Feasible Labeled Alignment $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ is a labeled alignment of X and Y where $\mathcal{L}(\mathcal{X})$ and $\mathcal{L}(\mathcal{Y})$ are feasible labeling. In Figure 1, (ii) and (iii) are two feasible labeled alignments of X and Y, with (iii) being one of minimum cost.

We are now ready to give the main optimization problem allowing to infer a most parsimonious history of duplications and losses leading to present-day genomes from a common ancestor.

Problem 1. ***Duplication-Loss Alignment****[DLA]*
Input*: Two genomes* X *and* Y*.*
Output*: A Feasible Labeled Alignment* $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ *of minimum cost.*

This problem has been shown NP-hard in [5]. An exact pseudo-boolean linear programming algorithm has been developed in [11] for this problem. The next section presents an alternative approach based on dynamic programming.

3 A Dynamic Programming Approach

Let $|X| = n$ and $|Y| = m$. Let $C(i,j)$ ($C^f(i,j)$ respectively) be the minimum cost of a labeled (feasible labeled respectively) alignment of two prefixes $X[1,i]$ and $Y[1,j]$ of X and Y. Then the problem is to compute $C^f(m,n)$. A natural approach sketched in [11] proceeds in two steps:

• Step 1. Unlabeled Alignment. Based on a dynamic programming approach, compute $C(i,j)$, for $1 \leq i \leq n$ and $1 \leq j \leq m$. Recurrences given in [11] allow to compute all values $M(i,j)$, $D_X(i,j)$, $D_Y(i,j)$, $L_X(i,j)$ and $L_Y(i,j)$ reflecting the minimum cost of an alignment $(\mathcal{X}_i, \mathcal{Y}_j)$ of $X[1,i]$ and $Y[1,j]$ satisfying respectively, the constraint that the last characters of $\mathcal{X}_i$ and $\mathcal{Y}_j$ represent a match, a duplication in $\mathcal{X}$ or in $\mathcal{Y}$, a loss in $\mathcal{X}$ or in $\mathcal{Y}$.

After computing all the values leading to $C(m,n)$, a bottom-up approach allows to output a labeled alignment $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ of minimum cost $C(m,n)$. Unfortunately, $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ is not necessarily a feasible alignment, as the recurrences for $D_X(i,j)$ may lead to invalid cyclic evolutionary scenarios. Notice that, as the DLA problem has been recently shown to be NP-complete [5], unless $P = NP$, no alternative recurrences would lead to a polynomial-time algorithm for computing $C^f(m,n)$.
• Step 2. Minimum Labeling Alignment. Consider an (unlabeled) alignment $(\mathcal{X}, \mathcal{Y})$ output by Step 1, and label it in an optimal way, e.g. find labeling $\mathcal{L}(\mathcal{X})$ and $\mathcal{L}(\mathcal{Y})$ for $\mathcal{X}$ and $\mathcal{Y}$ respectively, such that $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ is a feasible labeled alignment of minimum cost over all possible labeling of $(\mathcal{X}, \mathcal{Y})$. Notice that once the genomes are aligned, each labeling can be computed independently. Hence, the Minimum Labeling Alignment problem can be formulated as follows:

Problem 2. ***Minimum Labeling Alignment****[MLA]*
Input: *An aligned genome $\mathcal{X}$.*
Output: *A Feasible Labeling $\mathcal{L}(\mathcal{X})$ of minimum cost.*

The complexity of the MLA problem, as well as an appropriate algorithm to solve it, were left open in [11]. These are precisely the goals of our paper. It has to be noted that this approach cannot lead to an exact algorithm, as an alignment of minimum cost $C(m,n)$ does not necessarily lead to a feasible alignment of minimum cost $C^f(m,n)$. For example in Figure 1, an optimal labeling for alignment (i) of minimum cost $C(m,n) = 4$ leads to the feasible alignment (ii) of cost 5, which is not optimal, as (iii) is a better feasible alignment of cost 4.

Cost: As in [11], we will consider $c(D(k)) = 1$ and $c(L(k)) = k$. This leads to a natural weight of an evolutionary history in term of number of segmental duplications (duplication of a string of adjacent genes) and single losses (loss of a single gene). Although segmental deletions are also likely to occur during evolution, accumulation of mutations transforming a single gene into a pseudogene is the most frequent cause of gene loss. From an optimization point of view, the DLA problem is trivial if we count segmental losses as single events in the same way as duplications, that is $c(L(k)) = 1$. Indeed, in this case, a most parsimonious labeled alignment can always be obtained by ignoring duplications.

4 Hardness of Minimum Labeling Alignment

In this section, we prove that the MLA problem is APX-hard, even if each character (gene) has at most 5 occurrences in an aligned genome $\mathcal{X}$, by giving an L-reduction from the Minimum Vertex Cover problem on Cubic graphs (MVCC), known to be APX-hard [1], to MLA (for details on L-reduction see [2]). A graph is cubic iff each vertex of the graph has degree 3. Given a cubic graph $G = (V, E)$, with $V = \{v_1, \ldots, v_n\}$, MVCC asks for a minimum cardinality set $V' \subseteq V$, such that for each $\{v_i, v_j\} \in E$, at least one of v_i, v_j belongs to V'.

Next, we present the L-reduction from MVCC to MLA. Let $G = (V, E)$ be a cubic graph. Define the following ordering on the edges in E: $\{v_i, v_j\} < \{v_x, v_y\}$ if and only if $i < x$, or (in case $i = x$) $j < y$. Based on this ordering, we denote the edges incident on v_i, as the first, the second and the third edges of v_i. In what follows, given $v_i \in V$, we denote with $\{v_i, v_j\}$, $\{v_i, v_h\}$, $\{v_i, v_k\}$ the first, the second and the third edges respectively of G incident on v_i.

First, we define the aligned genome $\mathcal{X}$ corresponding to the cubic graph G. We present an overview of the construction of $\mathcal{X}$, then we give the details of the construction. The aligned genome $\mathcal{X}$ consists of two *parts* (see Fig. 2): the leftmost part is called the *Vertex-Edge-set Part* (VE-Part), the rightmost part is called the *Auxiliary Part* (A-Part). Each part is then divided into substrings, called *blocks*. Each position of $\mathcal{X}$ in the A-part is a match, while positions in the VE-part can be either matches or mismatches. Hence a labeling $\mathcal{L}$ of $\mathcal{X}$ is computed by labeling the mismatched positions in the VE-part of $\mathcal{X}$.

The VE-part of $\mathcal{X}$ consists of the concatenation of $|V| + |E|$ blocks (see Fig. 2). For each vertex $v_i \in V$ there is one block $B_{VE}(v_i)$ in the VE-part of $\mathcal{X}$; for each edge $\{v_i, v_j\} \in E$, there is one block $B_{VE}(e_{i,j})$ in the VE-part of $\mathcal{X}$.

The A-part of $\mathcal{X}$ consists of the concatenation of $2|V|$ blocks (see Fig. 2). For each $v_i \in V$, there exist two blocks $B_{A,1}(v_i)$, $B_{A,2}(v_i)$ in the A-part of $\mathcal{X}$. Now,

$$\mathcal{X} = \underbrace{B_{VE}(v_1) \ldots B_{VE}(v_n) B_{VE}(e_{1,a}) \ldots B_{VE}(e_{z,w})}_{\text{VE-part}} \cdot$$
$$\cdot \underbrace{B_{A,1}(v_1) B_{A,2}(v_1) \ldots B_{A,1}(v_n) B_{A,2}(v_n)}_{\text{A-part}}$$

Fig. 2. The structure of the aligned genome $\mathcal{X}$

we define the specific values of the blocks of $\mathcal{X}$. Given an edge $\{v_i, v_j\} \in E$, where $i < j$, $\{v_i, v_j\}$ is the p-th edge of v_i, $1 \leq p \leq 3$, and the q-th edge of v_j, $1 \leq q \leq 3$, we define its associated block $B_{VE}(e_{i,j})$ as follows:

$$B_{VE}(e_{i,j}) = s_{e,i,j} x_{i,p} e_{i,j,1} e_{i,j,2} x_{j,q}$$

where the first position of $B_{VE}(e_{i,j})$, that is the position containing character $s_{e,i,j}$, is a match and each other position of $B_{VE}(e_{i,j})$ is a mismatch.

Now, we define the block $B_{VE}(v_i)$, with $v_i \in V$. First, define the i-encoding of $\{v_i, v_j\}$, denoted as $i\text{-}enc_{i,j}$, as the following string: $i\text{-}enc_{i,j} = x_{i,p}e_{i,j,1}e_{i,j,2}$. Moreover, let $i\text{-}enc^l_{i,j} = x_{i,p}$, and $i\text{-}enc^r_{i,j} = e_{i,j,1}e_{i,j,2}$. The j-encoding of $\{v_i, v_j\}$, denoted as $j\text{-}enc_{i,j}$, is defined as follows: $j\text{-}enc_{i,j} = e_{i,j,1}e_{i,j,2}x_{j,q}$, and $j\text{-}enc^l_{i,j} = e_{i,j,1}e_{i,j,2}$, $j\text{-}enc^r_{i,j} = x_{j,q}$.

The block $B_{VE}(v_i)$ is defined as follows:

$$B_{VE}(v_i) = s_i z_{i,1} z_{i,2}\ i\text{-}enc_{i,j}\ z_{i,3} z_{i,4}\ i\text{-}enc_{i,h}\ z_{i,5} z_{i,6}\ i\text{-}enc_{i,k}\ z_{i,7} z_{i,8}$$

$B_{VE}(v_i)$ contains one matched position, the first position containing character s_i, and 17 mismatched positions (from position 2 to position 18 of $B_{VE}(v_i)$).

Now, we define the A-part of $\mathcal{X}$. Recall that each position of the A-part of $\mathcal{X}$ is a match. The block $B_{A,1}(v_i)$ is defined as follows:

$$B_{A,1}(v_i) = w_{i,1} z_{i,1} z_{i,2} w_{i,2} z_{i,3} z_{i,4} w_{i,3} z_{i,5} z_{i,6} w_{i,4} z_{i,7} z_{i,8}$$

The block $B_{A,2}(v_i)$ is defined as follows:

$$B_{A,2}(v_i) = u_{i,1} z_{i,2}\ i\text{-}enc^l_{i,j}\ u_{i,2}\ i\text{-}enc^r_{i,j}\ z_{i,3} u_{i,3} z_{i,4}\ i\text{-}enc^l_{i,h}\ u_{i,4}\ i\text{-}enc^r_{i,h}\ z_{i,5} \cdot$$

$$\cdot u_{i,5} z_{i,6}\ i\text{-}enc^l_{i,k}\ u_{i,6}\ i\text{-}enc^r_{i,k}\ z_{i,7}$$

Before giving the details of the proof, we give a high-level description of the reduction. We will show that each block $B_{VE}(v_i)$ can be labeled essentially in two possible ways (see Remark 4):

1. with a *type a labeling*, defining seven maximal duplications from substrings of blocks $B_{VE}(e_{i,j})$, $B_{VE}(e_{i,h})$, $B_{VE}(e_{i,k})$, $B_{A,1}(v_i)$ to substrings of block $B_{VE}(v_i)$; a *type a labeling* is the optimal labeling of $B_{VE}(v_i)$ (see Lemma 6) and has a cost of 7;
2. with a *type b labeling*, defining six maximal duplications from substrings of block $B_{A,2}(v_i)$ to substrings of block $B_{VE}(v_i)$ and two losses; a *type b labeling* is a suboptimal labeling of $B_{VE}(v_i)$ (see Lemma 6) and has a cost of 8.

Thanks to the property of block $B_{VE}(e_{i,j})$ (see Remark 5 and Lemma 7), we can relate these two kinds of labeling with a cover of G (see Lemma 8 and Lemma 9): a *type b labeling* of $B_{VE}(v_i)$ corresponds to a vertex v_i in a vertex cover V' of G, a *type a labeling* of $B_{VE}(v_i)$ corresponds to a vertex v_i in $V \setminus V'$ of G.

Now, we give the details of the reduction. First, we introduce some preliminaries properties of $\mathcal{X}$.

Remark 4. Consider a cubic graph $G = (V, E)$, and the corresponding instance $\mathcal{X}$ of MLA. Let v_i be a vertex of V, with $\{v_i, v_j\}$, $\{v_i, v_h\}$, $\{v_i, v_k\}$ the first, the second and the third edges of v_i respectively. A *type a labeling* of $B_{VE}(v_i)$ consists of the following 7 duplications:

– four duplications, each one from the substring $z_{i,2p-1}, z_{i,2p}$, $1 \le p \le 4$, of block $B_{A,1}(v_i)$, to the substring $z_{i,2p-1}, z_{i,2p}$, of block $B_{VE}(v_i)$;

– a duplication from the substring $i\text{-}enc_{i,x}$, with $x \in \{j, h, k\}$, of $B_{VE}(e_{ix})$ to the substring $i\text{-}enc_{i,x}$ of $B_{VE}(v_i)$.

A *type b labeling* labeling of $B_{VE}(v_i))$ consists of the following 6 duplications and 2 losses (hence it has a cost of 8):

– six duplications from substrings of $B_{A,2}(v_i)$ to substrings of $B_{VE}(v_i)$ (specifically for the six substrings $z_{i,2}$ $i\text{-}enc^l_{i,j}$, $i\text{-}enc^r_{i,j}$ $z_{i,3}$, $z_{i,4}$ $i\text{-}enc^l_{i,h}$, $i\text{-}enc^r_{i,h}$ $z_{i,5}$, $z_{i,6}$ $i\text{-}enc^l_{i,k}$, $i\text{-}enc^r_{i,k}$ $z_{i,7}$);
– two losses for the leftmost position and the rightmost position of $B_{VE}(v_i)$.

Notice that in a *type b labeling* for $B_{VE}(v_i)$, there is no duplication from substrings of $B_{VE}(e_{ij})$, $B_{VE}(e_{ih})$, $B_{VE}(e_{ik})$ to substrings of $B_{VE}(v_i)$.

Remark 5. Let $G = (V, E)$ be a cubic graph, let $\{v_i, v_j\} \in E$, with $i < j$, be the p-th edge of v_i, $1 \le p \le 3$, and the q-th edge of v_j, $1 \le q \le 3$. Let $\mathcal{X}$ be the corresponding instance of MLA. The following two labeling of $B_{VE}(e_{i,j})$ (recall that the first position of $B_{VE}(e_{i,j})$ is a match) have cost 2:

– one duplication from the substring $x_{i,p}e_{i,j,1}e_{i,j,2}$ of $B_{VE}(v_i)$ to the substring $x_{i,p}e_{i,j,1}e_{i,j,2}$ of $B_{VE}(e_{i,j})$, one loss for the last position of $B_{VE}(e_{i,j})$
– one duplication from the substring $e_{i,j,1}e_{i,j,2}x_{j,q}$ of $B_{VE}(v_j)$ to the substring $e_{i,j,1}e_{i,j,2}x_{j,q}$ $B_{VE}(e_{i,j})$, one loss for the second position of $B_{VE}(e_{i,j})$

Now, we are ready to show that a *type a labeling* is the only optimal labeling for $B_{VE}(v_j)$.

Lemma 6. *Let $G = (V, E)$ be an instance of MVCC and let $\mathcal{X}$ be the corresponding instance of MLA. Then, given a block $B_{VE}(v_i)$, with $v_i \in V$: (1) any feasible labeling of $B_{VE}(v_i)$ has a cost of at least 7; (2) if a labeling has cost of 7, then such a labeling is a* type a labeling *of $B_{VE}(v_i)$.*

Proof. (*Sketch.*) (1) The proof follows from a simple counting argument. Block $B_{VE}(v_i)$ contains 17 unmatched positions. By construction the leftmost position and the rightmost position of $B_{VE}(v_i)$ are labeled by duplications of length at most 2. The remaining positions are at least 13, and since by construction are labeled by duplications of length at most 3, it follows that at least $2 + \lceil \frac{13}{3} \rceil = 7$ duplications are required for each feasible labeling of $B_{VE}(v_i)$.

(2) It is easy to see that if a feasible labeling of $B_{VE}(v_i)$ contains only duplications from substrings of $B_{VE}(e_{i,j})$, $B_{VE}(e_{i,h})$, $B_{VE}(e_{i,k})$, $B_{A,1}(v_i)$, then it has a cost of 7 iff is a *type a labeling*. Similarly if a feasible labeling of $B_{VE}(v_i)$ contains only duplications from substrings of $B_{A,2}(v_i)$, it has a cost of at least 8. Assume that a feasible labeling $\mathcal{L}$ of $B_{VE}(v_i)$ contains a duplication $D = (\mathcal{X}[i_1, i_2], \mathcal{X}[j_1, j_2])$, where $\mathcal{X}[i_1, i_2]$ is a substring of $B_{A,2}(v_i)$, and a duplication from a substring of one of $B_{VE}(e_{i,j})$, $B_{VE}(e_{i,h})$, $B_{VE}(e_{i,k})$, $B_{A,1}(v_i)$. It is easy to see that D can (eventually) be extended so that it is a maximal duplication. Then by replacing each other duplication of $\mathcal{L}$ having as a target a substring of $B_{VE}(v_i)$ with a duplication from substrings of $B_{A,2}(v_i)$ (or a loss), we obtain a *type b labeling*. This implies that $\mathcal{L}$ has a cost of at least 8. □

Now, we prove a property on the labeling of a block $B_{VE}(e_{i,j})$.

Lemma 7. *Let $G = (V, E)$ be an instance of MVCC and let $\mathcal{X}$ be the corresponding instance of MLA. Then, each feasible labeling of $B_{VE}(e_{i,j})$, with $\{v_i, v_j\} \in E$, has a cost of at least 2, in which case $B_{VE}(e_{i,j})$ must be labeled with one duplication having target in $B_{VE}(v_i)$ or in $B_{VE}(v_j)$.*

Proof. By construction, since there is no other substring in the aligned genome $\mathcal{X}$ identical to $B_{VE}(e_{i,j})$, it follows that any labeling of $B_{VE}(e_{i,j})$ requires a cost of at least 2. Now, assume that $B_{VE}(e_{i,j})$ is not labeled by a duplication having a target in $B_{VE}(v_i)$ or in $B_{VE}(v_j)$. Then, by construction, either each position of $B_{VE}(e_{i,j})$ is labeled as a loss (the cost of such labeling is 4) or the position corresponding to the substring $e_{i,j,1}, e_{i,j,2}$ of $B_{VE}(e_{i,j})$ is labeled as a duplication from a substring of $B_{A,2}(v_i)$, implying a cost of 3 for the labeling. $\square$

Now, we are ready to prove the two main properties of the reduction in Lemma 8 and in Lemma 9.

Lemma 8. *Let G be an instance of MVCC and let $\mathcal{X}$ be the corresponding instance of MLA. Then, given a vertex cover $V' \subseteq V$ of G, we can compute in polynomial time a solution of MLA over instance $\mathcal{X}$ of cost $8|V'|+7|V \setminus V'|+2|E|$.*

Proof. (*Sketch*). Given a cover V' of G, we define a solution of MLA over instance $\mathcal{X}$ having cost $8|V'| + 7|V \setminus V'| + 2|E|$ as follows: (1) for each $v_i \in V'$, define a *type b labeling* for the corresponding block $B_{VE}(v_i)$ (of cost of 8, see Remark 4); (2) for each $v_i \in V \setminus V'$, define a *type a labeling* for the corresponding block $B_{VE}(v_i)$ (of cost of 7, see Remark 4); (3) a duplication of cost 2 for each $B_{VE}(e_{i,j})$ associated with edge $\{v_i, v_j\} \in E$ (see Remark 5). Since V' is a vertex cover of G, at least one of $v_i, v_j \in V'$, hence this labeling is feasible. $\square$

Lemma 9. *Let G be an instance of MVCC and let $\mathcal{X}$ be the corresponding instance of MLA. Then, given a feasible labeling of $\mathcal{X}$ of cost $8p+7(|V|-p)+2|E|$, we can compute in polynomial time a vertex cover of G of size at most p.*

Proof. (*Sketch*). Let $\mathcal{L}$ be a feasible labeling of $\mathcal{X}$ of cost $8p + 7(|V| - p) + 2|E|$. First, by Lemma 6, we can assume that $B_{VE}(v_i)$ is associated in $\mathcal{L}$ either with a *type a labeling* or with a *type b labeling.*

Now, consider a block $B_{VE}(e_{i,j})$, with $\{v_i, v_j\} \in E$. We show that we can assume that at least one of $B_{VE}(v_i)$, $B_{VE}(v_j)$ has a *type b labeling* in $\mathcal{L}$. If this is not the case, $B_{VE}(e_{i,j})$ cannot be labeled with a duplication having source in $B_{VE}(v_i)$, $B_{VE}(v_j)$, hence by Lemma 7, the cost of the labeling of $B_{VE}(e_{i,j})$ is at least 3. We compute in polynomial time a feasible labeling $\mathcal{L}'$, such that $c(\mathcal{L}') \leq c(\mathcal{L})$, as follows: (1) define a *type b labeling* for one of $B_{VE}(v_i)$, $B_{VE}(v_j)$, w.l.o.g. $B_{VE}(v_i)$; (2) define a duplication D from the substring $i\text{-}enc_{i,j}$ of $B_{VE}(v_i)$ to the substring $i\text{-}enc_{i,j}$ of $B_{VE}(e_{i,j})$, and a loss for the unmatched position of $B_{VE}(e_{i,j})$ not contained in the target of D. Hence we can assume that, for each block $B_{VE}(e_{i,j})$, at least one of $B_{VE}(v_i)$, $B_{VE}(v_j)$ has a *type b labeling* in $\mathcal{L}$. It follows that we can define a vertex cover V' of G as follows: $V' = \{v_i : B_{VE}(v_i)$ has a *type b labeling* in $\mathcal{L}\}$. Since the cost of $\mathcal{L}$ is at most $8p + 7(|V| - p) + 2|E|$, it follows that $|V'| \leq p$. $\square$

The following result is a direct consequence of Lemmas 8 and 9.

Theorem 1. *MLA is APX-hard.*

5 An Efficient Heuristic

We now present DLAlign , which is a heuristic based on the dynamic programming approach (Section 3) for the Duplication-Loss Alignment (DLA, Problem 1) of two genomes X and Y. Recall that $|X| = n$ and $|Y| = m$.

• **Step 1.** Dynamic Programming:
– Compute all the values of $C(i,j)$, for $1 \leq i \leq n$ and $1 \leq j \leq m$;
– Output a labeled alignment $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$ of cost $C(m,n)$. To limit the possibility of creating cycles we do the following: (i) in the bottom-up approach used to output a labeled alignment after filling the dynamic programming table C, we choose a match operation whenever possible; (ii) for any duplication involving a given string Z, the rightmost position of Z in the genome is always chosen to be the source of the duplication.
• **Step 2.** Minimum Labeling Alignment: Resolve each duplication cycle $\mathcal{D}$ of $(\mathcal{L}(\mathcal{X}), \mathcal{L}(\mathcal{Y}))$, by interpreting the shortest overlapping string of $\mathcal{D}$ as a loss rather than a duplication (see Examples (i) and (ii) in Figure 1).

Complexity: For simplicity, suppose $|X| = |Y| = n$. From the recurrences detailed in [11], each $C(i,j)$, for $1 \leq i,j \leq n$, can be computed in time $O(n)$ which leads to an $O(n^3)$ algorithm for Step 1. As for Step 2, it requires constructing a graph for X (Y respectively): for each duplication, add two vertices corresponding to its source and target, and one edge from source to target. Constructing the graphs, findings the cycles and resolving them can be done in time $O(n^3)$, which leads to an $O(n^3)$ worst-time complexity for the whole heuristic.

5.1 Simulations

A random string R was drawn from the set of all strings of length n on an alphabet of size a, and l moves were then applied to R to obtain an ancestral genome A. To obtain the extant genomes X and Y, l more moves were applied to A for each. The set of moves were segmental duplications and single gene losses. The length of a duplication was drawn from a Gaussian distribution with mean 5 and standard deviation 2; these lengths were consistent with those observed for the tRNA repertoire in *Bacillus* lineages [11].

Execution time: With $2l/n = 1/5$ and $a/n = 1/2$, statistics similar to those observed for the tRNA repertoire in *Bacillus*, strings of length 5000 took a couple of days to be processed by the linear programming algorithm on a standard PC workstation with 4 GB of memory. In comparison, the same data have been processed by DLAlign on the same computer in less than two seconds.

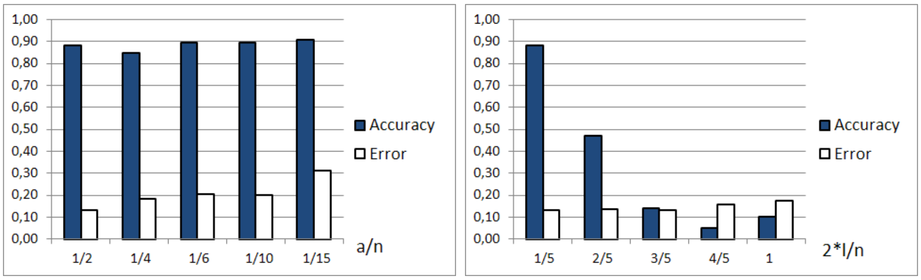


Fig. 3. The score returned by DLAlign compared to the optimal one returned by the linear programming algorithm, for datasets of size up to $n = 200$. The left diagram is obtained by varying the alphabet size a (x-axis is a/n), and the right diagram by varying the number of moves l (x-axis is $2 * l/n$). See text for more details.

Accuracy: We compare *Res*, the alignment cost returned by DLAlign, with the optimal cost *Opt* obtained by running the linear programming algorithm. Due to the exponential-time complexity of the later, we had to restrict ourselves to relatively small values of n, a and l. Results of Figure 3 are averaged over up to $Total = 1000$ simulations. White bars refer to $Error = \frac{Res-Opt}{Res}$, and blue ones to $Accuracy = \frac{NbOpt}{Total}$, where *NbOpt* is the number of simulations among *Total* for which DLAlign outputs the optimal alignment (i.e. $Error = 0$).

With ratios $2l/n = 1/5$ and $a/n = 1/2$, DLAlign returns the optimal alignment cost for more than 85% of the simulations. This accuracy rate remains stable for decreasing alphabet size, i.e. increasing number of gene copies (left diagram in Figure 3), but quickly drops with increasing number l of moves (right diagram). Notice however that, even for a number of moves being equal to the size of the strings, the error rate *Error* always remains lower than 0.16.

6 Conclusion

In this paper, we investigated the problem of aligning two genomes, based on a duplication and loss model of evolution. We developed a heuristic in two steps: first use dynamic programming to output a best candidate solution, then consider MLA to compute a feasible solution. The heuristic exhibited a high degree of accuracy on simulated datasets. Moreover, it is a thousands of times faster than the previously developed linear programming algorithm, which makes possible its application to large genomes, and allows generalization to multiple genome alignment in a phylogenetic context. From a theoretical point of view, we showed that the MLA problem is APX-hard even when each gene has at most five occurrences in a genome. Interesting future work will be to investigate the approximation and parametrized complexity of MLA.

Acknowledgements. We thank Krister M. Swenson for advices and help with simulations.

References

1. Alimonti, P., Kann, V.: Some APX-completeness results for cubic graphs. Theoretical Computer Science 237(1-2), 123–134 (2000)
2. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M.: Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties. Springer, Heidelberg (1999)
3. Bergeron, A.: A Very Elementary Presentation of the Hannenhalli-Pevzner Theory. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 106–117. Springer, Heidelberg (2001)
4. Bourque, G., Pevzner, P.: Genome-scale evolution: Reconstructing gene orders in the ancestral species. Genome Research 12, 26–36 (2002)
5. Canzar, S., Andreotti, S.: A branch-and-cut algorithm for the 2-species duplication-loss phylogeny problem. CoRR abs/1208.2698 (2012)
6. El-Mabrouk, N.: Genome rearrangement with gene families. In: Mathematics of Evolution and Phylogeny, pp. 291–320. Oxford University Press, Oxford (2005)
7. El-Mabrouk, N., Sankoff, D.: Analysis of Gene Order Evolution beyond Single-Copy Genes. In: Evolutionary Genomics: Statistical and Computational Methods. Methods in Molecular Biology. Springer (Humana), New York (2012)
8. El-Mabrouk, N.: Genome Rearrangement by Reversals and Insertions/Deletions of Contiguous Segments. In: Giancarlo, R., Sankoff, D. (eds.) CPM 2000. LNCS, vol. 1848, pp. 222–234. Springer, Heidelberg (2000)
9. Fertin, G., Labarre, A., Rusu, I., Tannier, E., Vialette, S.: Combinatorics of genome rearrangements. The MIT Press, Cambridge (2009)
10. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). Journal of the ACM 48, 1–27 (1999)
11. Holloway, P., Swenson, K., Ardell, D., El-Mabrouk, N.: Evolution of Genome Organization by Duplication and Loss: An Alignment Approach. In: Chor, B. (ed.) RECOMB 2012. LNCS, vol. 7262, pp. 94–112. Springer, Heidelberg (2012)
12. Ma, J., Zhang, L., Suh, B., Raney, B., Burhans, R., Kent, W., Blanchette, M., Haussler, D., Miller, W.: Reconstructing contiguous regions of an ancestral genome. Genome Research 16, 1557–1565 (2007)
13. Marron, M., Swenson, K.M., Moret, B.M.E.: Genomic Distances Under Deletions and Insertions. In: Warnow, T.J., Zhu, B. (eds.) COCOON 2003. LNCS, vol. 2697, pp. 537–547. Springer, Heidelberg (2003)
14. Moret, B., Wang, L., Warnow, T., Wyman, S.: New approaches for reconstructing phylogenies from gene order data. Bioinformatics 173, S165–S173 (2001)
15. Sankoff, D., Blanchette, M.: The Median Problem for Breakpoints in Comparative Genomics. In: Jiang, T., Lee, D.T. (eds.) COCOON 1997. LNCS, vol. 1276, pp. 251–264. Springer, Heidelberg (1997)

Maximizing Entropy over Markov Processes*

Fabrizio Biondi[1], Axel Legay[2], Bo Friis Nielsen[3], and Andrzej Wąsowski[1]

[1] IT University of Copenhagen, Denmark
{fbio,wasowski}@itu.dk
[2] INRIA Rennes, France
axel.legay@inria.fr
[3] Technical University of Denmark, Lyngby, Denmark
bfn@imm.dtu.dk

Abstract. The channel capacity of a deterministic system with confidential data is an upper bound on the amount of bits of data an attacker can learn from the system. We encode all possible attacks to a system using a probabilistic specification, an Interval Markov Chain. Then the channel capacity computation reduces to finding a model of a specification with highest entropy.

Entropy maximization for probabilistic process specifications has not been studied before, even though it is well known in Bayesian inference for discrete distributions. We give a characterization of global entropy of a process as a reward function, a polynomial algorithm to verify the existence of an system maximizing entropy among those respecting a specification, a procedure for the maximization of reward functions over Interval Markov Chains and its application to synthesize an implementation maximizing entropy.

We show how to use Interval Markov Chains to model abstractions of deterministic systems with confidential data, and use the above results to compute their channel capacity. These results are a foundation for ongoing work on computing channel capacity for abstractions of programs derived from code.

1 Introduction

Quantified Information Flow [7] is a quantitative approach to compute the number of bits of information an attacker would gain about the confidential data of a system by interacting with the system and observing its behavior.

Leakage is defined as the difference between the attacker's information [17] about the confidential data before and after the attack. If we view the system as a channel through which the attacker gets information about the secret, its *capacity* can be computed as the maximum leakage over all prior infromations of attackers. This provides a security guarantee for the system, as no attack can leak an amount of information higher than the system's channel capacity [15]. For a deterministic system, the leakage is the entropy of the observable behavior of the system [13], and thus computing the channel capacity reduces to computing the behavior of the system that maximizes entropy.

Our goal is to develop theories and algorithms to synthesize the process with maximum entropy among all those respecting a given probabilistic specification, allowing

* The research presented in this paper has been partially supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology.

us to give a security guarantee valid for all the infinite processes respecting the specification. We use Markov chains (MCs) as process models and Interval Markov Chains (Interval MCs) [11] as specification models. We use the continuity of the real-valued intervals to encode the infinite number of possible attackers and implementations we want to consider.

In the theoretical sense, in this paper we extend the well know Maximum Entropy Principle of Jaynes [10], from constraints on probability distributions to interval constraints on discrete probabilistic processes. We consider Interval MCs as constraints over probabilistic processes and resolve them for maximum entropy. As a result we validate the intuition that channel capacity computation as known in security research corresponds to obtaining least biased solutions in Bayesian inference for processes.

Given a deterministic protocol specified as an Interval MC, we show how to:

1. *Compute the entropy of a given implementation.* We provide a polynomial-time procedure to compute entropy for a Markov chain, by reduction to computation of the Expected Total Reward [16, Chpt. 5] of a local non-negative reward function associated with states over the infinite horizon.
2. *Check if a protocol allows for insecure implementations.* We provide a polynomial-time procedure for deciding finiteness and boundedness of the entropy of all implementations of an Interval MC. In general a Maximum Entropy implementation might not exist. Implementations might be non-terminating and accumulate infinite entropy, or they may have arbitrarily high, i.e. unbounded, finite entropy, so standard optimization techniques would diverge. In such case it is not possible to give a security guarantee for the implementations. We provide a polynomial-time algorithm to distinguish the two cases. If a protocol allows implementations with infinite or unbounded entropy then no matter the size n of the secret, it will have an implementation leaking all n bits of it. We detect this so that the designer can strengthen the protocol design appropriately.
3. *Compute channel capacity of a protocol.* This is a multidimensional nonlinear maximization problem on convex sets [18]. We use a numerical procedure for synthesizing with arbitrary approximation an implementation maximizing a reward function over Interval MCs. An Interval MC can be considered as an infinite set of processes, and since entropy is a nonlinear function of all possible behaviors of a system, finding the one with highest entropy is not trivial. We apply this procedure to synthesize a Maximum Entropy process implementing an Interval MC; the entropy of such process is the channel capacity of all processes implementing the Interval MC.

Motivating Examples. Consider two examples of models of deterministic authentication processes. Figure 1a presents a specification of a two-step authentication protocol. A user is requested to input a username, and is rejected if the username is unknown. If it is correct the user is asked to input a password, and is accepted if the password corresponds to the username and rejected otherwise. The actual transition probabilities will depend on how many usernames exist in the system, on the respective passwords, on their length and on the attacker's knowledge about all of these. Staying at the specification level allows us to consider the worst case of all these possible combinations,

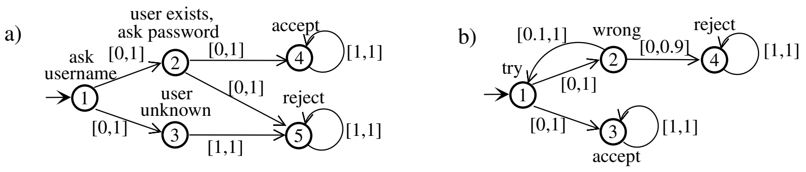


Fig. 1. a) Two-step Authentication b) Repeated Authentication

and thus gives an upper bound on the leakage. The Maximum Entropy implementation for the Two-step Authentication is given in Fig. 2. Its entropy is the channel capacity of the system over all possible prior informations and behaviors of the attacker and design choices.

Consider another example. Figure 1b presents an Interval MC specification of the Repeated Authentication protocol. A user inserts a password to authenticate, and is allowed access if the password is correct. If not, the system verifies if the password entered is in a known black list of common passwords, in which case it rejects the user, considering it a malicious attacker. If the password is wrong but not black listed the user is allowed to try again. The black list cannot cover more than 90% of the possible selection of passwords.

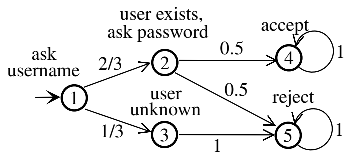


Fig. 2. Maximum Entropy implementation for the Two-step Authentication

Note that the transition probabilities from state 2 depend on a design choice left to the implementer of the system and on the attacker's knowledge about it, while the transition probabilities from state 1 depend on the length of the password and the attacker's knowledge about it. By abstracting these different sources of nondeterminism at the same time we maximize entropy over all possible combinations of design choices and attackers, effectively finding the channel capacity of the specification.

Implementations with higher entropy reveal more information about the system's secret. This is consistent with our intuition. For instance, in the example in Fig 1b decreasing the black list, which decreases probability of reject, increases the possible information leakage. If the black list is empty the user can continue guessing the password indefinitely: the probability of eventually reaching state 3 is 1. In this implementation sooner or later the attacker will discover the password, and thus the system's secret will be completely revealed. So, a larger black list increases the chance that the system will enter the absorbing state 4 and leak less information, and symmetrically a smaller black list increases the leakage.

In fact, it is not possible to give a Maximum Entropy implementation for such protocol. Whatever is the length n of the secret, it is possible to give an implementation that leaks all n bits of information. We show how these undesirable cases can be recognized in polynomial time and how the specification can be modified to avoid them.

Figure 3bc shows two implementations of the Repeated Authentication presented in Fig. 3a. None of them maximizes entropy. In fact, it is not possible to give a

Maximum Entropy implementation for such protocol. We will discuss the significance of this in Sect. 6. The Maximum Entropy implementation for the Two-step Authentication is given in Fig. 2. We explain how it has been synthesized in Sect. 5.

Related Work. Channel capacity as a security guarantee [15] has been studied for many different models of computation. Chatzikokolakis, Palamidessi and Panangaden use it to give a formula for anonymity analysis of protocols described by weakly symmetric matrices [5], based on the probabilistic anonymity approach by Bhargava and Palamidessi [2]. Chen and Malacaria generalize this result to asymmetric protocols [6] and also study the channel capacity of deterministic systems under different observation models [14]. Our method, unlike theirs, handles infinite classes of models instead of single models, and uses different states of a Markov chain to represent the different logical states of the system instead of considering the system as a function from inputs to outputs.

2 Background on Probabilistic Processes

Definition 1. *A triple $\mathcal{C} = (S, s_0, P)$ is a Markov Chain (MC), if S is a finite set of states containing the initial state s_0 and P is an $|S| \times |S|$ probability transition matrix, so $\forall s, t \in S.\ P_{s,t} \geq 0$ and $\forall s \in S.\ \sum_{t \in S} P_{s,t} = 1$.*

We slightly abuse the notation, interpreting states as natural numbers and indexing matrices with state names. This is reflected in figures by labeling of states both with textual descriptions and numbers.

A state is *deterministic* if it has exactly one outgoing transition with probability 1, *stochastic* otherwise. It is known [8] that the probability of transitioning from any state s to a state t in k steps can be found as the entry of index (s,t) in P^k. We call $\pi^{(k)}$ the probability distribution vector over S at time k and $\pi_s^{(k)}$ the probability of visiting the state s at time k; note that $\pi^{(k)} = \pi_0 P^k$, where $\pi_s^{(0)}$ is 1 if $s = s_0$ and 0 otherwise. A state t is *reachable* from a state s if $\exists k. P^k_{s,t} > 0$. We assume that all states are reachable from s_0 in the MCs considered. A subset $R \subseteq S$ is *strongly connected* if for each pair of states $s, t \in R$, t is reachable from s. Let ξ_s denote the *residence time* in a state s: $\xi_s = \sum_{n=0}^{\infty} P^n_{s_0,s}$.

For the purpose of defining entropy, it is useful to consider the alternative, less automata-theoretical but more probabilistic, view of an MC. An MC can be seen as an infinite sequence of discrete random variables $(X_n, n \in \mathbb{N})$, where $\mathbf{P}(X_k = s) = \pi_s^{(k)}$ represents the probability that the chain will be visiting state $s \in S$ at time k. The processes must respect the *Markov property*: $P(X_n = s_n \mid X_{n-1} = s_{n-1}, \ldots, X_0 = s_0) = P(X_n = s_n \mid X_{n-1} = s_{n-1}), \forall s_0, s_1, \ldots, s_n \in S, n \in \mathbb{N}$.

A state s is *recurrent* iff $\xi_s = \infty$, *transient* otherwise. Residence time of each state of an MC can be calculated in polynomial time [16].

Usage of Markov chains to model generic secret-dependent processes has been previously introduced by the authors [3], including ways to automatically generate them from imperative program code. Each state of the MC represents a reachable combination of values of the public variables of the system and levels of knowledge about the private variables. We refer to [3] for the full discussion.

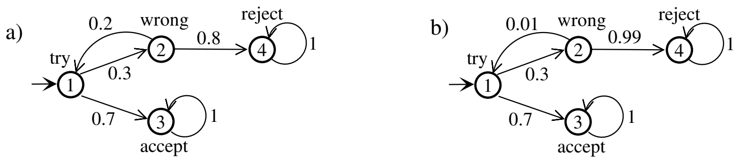


Fig. 3. a) Correct implementation of the Repeated Authentication. b) Incorrect implementation of the Repeated Authentication.

Definition 2. [4] *A closed-interval* Interval Markov Chain *(Interval MC) is a tuple* $\mathcal{I} = (S, s_0, \check{P}, \hat{P})$ *where* S *is a finite set of states containing the initial state* s_0*,* $\check{P}$ *is an* $|S| \times |S|$ *bottom transition probability matrix,* $\hat{P}$ *is a* $|S| \times |S|$ *top transition probability matrix, such that for each pair of states* $s, t \in S$ *we have* $\check{P}_{s,t} \leq \hat{P}_{s,t}$*.*

The following defines when an MC implements an Interval MC in the Uncertain Markov Chain (UMC) semantics [4]:

Definition 3. *A Markov chain* $\mathcal{C} = (S, s_0, P)$ implements *an Interval Markov Chain* $\mathcal{I} = (S, s_0, \check{P}, \hat{P})$*, written* $\mathcal{C} \vDash \mathcal{I}$*, if* $\forall s, t \in S.\ \check{P}_{s,t} \leq P_{s,t} \leq \hat{P}_{s,t}$*.*

An example of an Interval MC is the Repeated Authentication of Fig. 1b. The MC in Fig. 3a uses a black list with 80% of the passwords and is thus an implementation of the Interval MC, while the Markov chain in Figure 3b uses a black list with 99% and it is not an implementation of the Interval MC.

We assume that our Interval MCs are *coherent*, meaning that every value for each transition interval is attained by some implementation. Coherence can be established by checking that both following conditions hold [12]:
1) $\forall s, t \in S. \check{P}_{s,t} \geq (1 - \sum_{u \neq t} \hat{P}_{s,u})$ 2) $\forall s, t \in S. \hat{P}_{s,t} \leq (1 - \sum_{u \neq t} \check{P}_{s,u})$

Assuming coherence is not a limitation. If an Interval MC $\mathcal{I} = (S, s_0, \check{P}, \hat{P})$ is not coherent it can be made coherent in polynomial time [12]; we produce the coherent Interval MC $\mathcal{I}' = (S, s_0, \check{P}', \hat{P}')$ by changing the top and bottom transition probability matrices to the following:
1) $\check{P}'_{s,t} = \max(\check{P}_{s,t}, 1 - \sum_{u \neq t} \hat{P}_{s,u})$ 2) $\hat{P}'_{s,t} = \min(\hat{P}_{s,t}, 1 - \sum_{u \neq t} \check{P}_{s,u})$

The resulting coherent Interval MC $\mathcal{I}'$ is unique and has the same implementations as the original incoherent Interval MC $\mathcal{I}$ [12], so in particular it has an implementation iff $\mathcal{I}$ has at least one implementation.

A state s of an Interval MC is *deterministic* if $\exists t. \check{P}_{s,t} = 1$, *stochastic* otherwise. We say that a state t is *reachable* from a state s if $\exists s_1, s_2, ..., s_n \in S. s_1 = s \wedge s_n = t \wedge \hat{P}_{s_i, s_{i+1}} > 0$ for $1 \leq i < n$. We say that a subset $R \subseteq S$ is *strongly connected* if $\forall s, t \in R.\ t$ is reachable from s.

Note that if there is an implementation in which a subset of states $R \subseteq S$ is strongly connected, then R must be strongly connected in the Interval MC.

We often refer to deterministic, stochastic and nondeterministic behavior. We use the adjective *deterministic* for a completely predictable behavior, *stochastic* for a behavior that follows a probability distribution over some possible choices, and *nondeterministic* for a choice where no probability distribution is given.

3 Entropy of Processes and Specifications

The entropy of a discrete probability distribution quantifies lack of information about the events involved. This idea can be extended to quantify nondeterminism, understood as degree of unpredictability of an MC. For a discrete set of n events $(x_1, ..., x_n)$ *entropy* is defined as $-\sum_{i=1}^{n} \mathbf{P}(x_i) \log_2(\mathbf{P}(x_i))$ and is maximal for the uniform distribution, in which case its value is $\log_2 n$ [8]. For MCs, entropy is maximum for the process in which all possible paths in the chain have the same probability.

To define the entropy of a Markov chain $\mathcal{C}$ we need to introduce the concepts of *conditional entropy* and *joint entropy* [17,8]. Conditional entropy quantifies the remaining entropy of a variable Y given that the value of other random variables (X_i here) is known.

$$H(Y \mid X_1, \ldots, X_n) = -\sum_{t \in S} \sum_{s_1 \in S} \cdots \sum_{s_n \in S} \mathbf{P}(Y = t, X_1 = s_1, \ldots, X_n = s_n) \cdot \\ \cdot \log_2 \mathbf{P}(Y = t \mid X_1 = s_1, \ldots, X_n = s_n) \ ,$$

where $\mathbf{P}(Y = t, X_1 = s_1, \ldots, X_n = s_n)$ denotes the joint probability of the events $Y = t, X_1 = s_1, \ldots, X_n = s_n$.

Joint entropy is simply the entropy of several random variables computed jointly, i.e. the combined uncertainty due to the ignorance of n random variable. It turns out [8] that joint entropy can be calculated in the following way, using conditional entropy, which will be instrumental in our developments for MCs.

$$H(X_0, X_1, \ldots, X_n) = -\sum_{s_0 \in S} \sum_{s_1 \in S} \cdots \sum_{s_n \in S} \mathbf{P}(X_0 = s_0, X_1 = s_1, \ldots, X_n = s_n) \cdot \\ \cdot \log_2(\mathbf{P}(X_0 = s_0, X_1 = s_1, \ldots, X_n = s_n)) = \\ = H(X_0) + H(X_1 \mid X_0) + \cdots + H(X_n \mid X_0, X_1, \ldots, X_{n-1})$$

Now, the definition of entropy of an MC is unsurprising, if we take the view of the processes as a series of random variables; it is the joint entropy of these variables (recall that due to the Markov property, the automata-view, and the probabilistic view of MCs are interchangeable):

Definition 4. *We define the entropy of a Markov chain $\mathcal{C} = (X_n, n \in \mathbb{N})$ as the joint entropy over all*

$$X_n\text{:}H(\mathcal{C}) = H(X_0, X_1, X_2, \ldots) = \sum_{i=0}^{\infty} H\left(X_i \left| X_{i-1} \ldots X_0\right.\right).$$

Note that since we have assumed a single starting state in each MC, it is always the case that $H(X_0) = 0$. Also the above series always converges to a real number, or to infinity, since it is a sum of non-negative real numbers.

In leakage analysis, entropy corresponds to the information leakage of the system only when the system is deterministic and the attacker cannot interact with it [13]. Using probability intervals we can lift the latter restriction, as different distributions on the attacker's input would only lead to different transition probabilities, and the intervals already consider all possible transition probabilities.

The entropy of an MC is in general infinite; we will give a characterization in Corollary 6 in the next Section showing that the entropy of an MC is finite if and only if the chain is absorbing. Considering only absorbing MCs avoids the problem of the entropy of an MC being in general infinite. We always consider terminating protocols, and they can be encoded as absorbing MCs where the absorbing states represent the termination of the protocol; consequently the entropy of a Markov chain encoding a terminating process is always finite.

We stress that it is common [17,9] to compute the average entropy of each step of the MC and to call it the entropy of the MC, while it's technically an entropy rate [8]. Even though entropy rate is always finite, we want to compute the actual entropy since it represents the information leakage in a security scenario where the states of the MC are the observables of a deterministic program.

An alternative characterization of entropy of a process depends explicitly on which states get visited during the lifetime of the process. Since every state s has a probability distribution over the next state we can compute the entropy of that distribution, which we will call *local entropy* $L(s)$ of s: $L(s) = H(X_{k+1} \mid X_k = s) = -\sum_{t \in S} P_{s,t} \log_2 P_{s,t}$. Note that $L(s) \leq \log_2(|S|)$. Also the value of entropy of an MC is in general not equal to the sum of the local entropy values for each state. Such sum will have to be weighted against the residence time of each state to characterize the "global" entropy.

Now consider the Interval MC specification of the Repeated Authentication in Fig. 1b. Different implementations of it, like the ones in Fig. 3ab, will have different entropy values. We define a *Maximum Entropy implementation* for an Interval MC, as an implementation MC, which has entropy not smaller than entropy of any other implementation (if such exists). The boundedness and synthesis of the Maximum Entropy implementation of an Interval MC will be treated in Sect. 5. It may be that the maximum entropy is actually infinite, or that the set of attainable entropies is unbounded; we discuss these cases in Sect. 6.

4 Computing Entropy of Markov Chains

We now provide an algorithm for computing entropy of a given MC. We cast entropy as a non-negative reward function on an MC, and then apply standard techniques to compute it. We also provide a simple decision procedure for deciding whether entropy of an MC is finite.

A *non-negative reward function* over the transitions of an MC is a function $R : S \times S \to \mathbb{R}^+$ assigning a non-negative real value, called reward, to each transition. Given a reward function R we can compute the value of the reward for a concrete execution of an MC by summing reward values for the transitions exercised in the execution. More interestingly, we can compute the expected reward of each state $s \in S$ as $R(s) = \sum_{t \in S} P_{s,t} R_{s,t}$, and then the expected reward over the infinite behavior of an MC $\mathcal{C}$ is $R(\mathcal{C}) = \sum_{s \in S} R(s)\xi_s$ [16, Chpt. 5].

Each $R(s)$ can be computed in time linear in the number of states, so calculation of expected rewards for all states can be done in quadratic time. Since computing the residence time for a state s is in PTIME, we can also compute $R(\mathcal{C})$ in polynomial time.

Let the reward function be $R(s,t) = -\log_2 P_{s,t}$. Then the expected reward for each state is its local entropy, or $R(s) = -\sum_{t\in S} P_{s,t}\log_2 P_{s,t} = L(s)$. Note that this is an unorthodox non-negative reward function, since it depends on the choice of probability distribution, as a function of the form $R : S \times S \times P \to \mathbb{R}^+$. It turns out that the (global) entropy of the MC is the expected reward with this reward:

Theorem 5. *For an MC $\mathcal{C} = (S, s_0, P)$ we have that $H(\mathcal{C}) = \sum_{s\in S} L(s)\xi_s$*

As any other reward of this kind, the entropy of an MC can be infinite. Intuitively, the entropy is finite if it almost surely stops increasing. This happens if the execution is eventually confined to a set of states with zero local entropy (deterministic). Since the recurrent states of a chain are exactly the ones that are visited infinitely often, we obtain the following characterization:

Corollary 6. *The entropy $H(\mathcal{C})$ of a chain $\mathcal{C}$ is finite iff the local entropy of all its recurrent states is zero.*

The above observation gives us an algorithmic characterization of finiteness of entropy for MCs: the entropy of a chain is finite if and only if the chain has one or more absorbing states or absorbs into closed deterministic cycles. Entropy can only be infinite for infinite behaviors; for the first n execution steps the entropy is bounded by $n\log_2|S|$.

We can classify the processes in two categories: those which eventually terminate the stochastic behavior, and those which do not. Many processes become deterministic (or even terminate) after some time. This is the case for a terminating algorithm like a randomized primality test, or for randomized IP negotiation protocols like Zeroconf, which stops behaving randomly as soon as an IP number is assigned. Such processes have finite entropy. On the other hand, the processes that take probabilistic choices forever and never become deterministic have infinite entropy. Using Corollary 6 we characterize the processes having finite entropy as terminating and the processes having infinite entropy as non-terminating.

5 Maximum Entropy Implementation of an Interval MC

Interval MCs describe infinite sets of MCs. We now show how to find an implementation that maximizes entropy. Since our Markov chains represent the behavior of deterministic processes, the Maximum Entropy implementation we synthesize is also the one with maximum leakage, and its leakage is thus the channel capacity of all implementations.

In Fig. 2 we show the Maximum Entropy implementation of the Two-step Authentication specification in Fig. 1a. Its entropy of $\log_2 3 \approx 1.58496$ bits. This allows us to guarantee that none of the infinite possible implementations of the Two-step Authentication will leak more than $\log_2 3$ bits of information to any possible attacker.

Obtaining such an implementation is a challenging problem. In the first place, such an implementation may not exist, so we need an algorithm to verify its existence. Secondly, even if it exists finding it consist in solving a nonlinear optimization problem with constraints over an infinite domain.

In this section we present a new algorithm that given an Interval MC $\mathcal{I}$ finds its implementation, in the sense of Def. 3, that maximizes the entropy value. We propose a numerical approach to the general problem of solving Interval MCs for non-negative reward functions, and apply it to finding a Maximum Entropy implementation. We first check that such an implementation exists, and then proceed to synthesize it. Remember that an implementation maximizing the reward function $R(s,t) = -\log_2(P_{s,t})$ is a Maximum Entropy implementation.

The expected reward of a non-negative reward function may be infinite. An Interval MC admits implementations with infinite entropy if it has a state that can be recurrent and stochastic in the same implementation. We call this the *infinite* case.

If an Interval MC has a state that is recurrent in some implementations and stochastic in some others, but never both recurrent and stochastic in the same implementation, the set of entropies of its implementations is unbounded, despite all the individual implementations having finite entropy; an example is the Repeated Authentication in Fig. 1b. We call this the *unbounded* case. This happens because the reward assigned to a transition is not a constant, but a logarithmic function of the actual transition probability—the logarithm is taken of the value that depends on the probability distribution of the implementation. With such reward it is possible that the total reward value can be unbounded across possible implementations (not just finite or infinite as for classical non-negative rewards). Note that this does not happen with constant rewards, and is specific to our problem.

5.1 Existence of a Maximum Entropy Implementation

We now show an algorithm for determining whether an Interval MC has a Maximum Entropy implementation with finite entropy. To do this we first give a definition of end components [1] for Interval MCs. Then we show the algorithm for deciding the existence of a Maximum Entropy implementation.

We propose a definition of an end component for Interval MCs. An end component is a set of states of the Interval MC for which there exists an implementation such that once the behavior enters the end component it will stay inside it forever and choose all transitions inside it an infinite number of times with probability 1. We refer to [1] for further discussion.

For an Interval MC $\mathcal{I} = (S, s_0, \check{P}, \hat{P})$, $R \subseteq S$ is an end component of $\mathcal{I}$ then there is an implementation of $\mathcal{I}$ in which $\mathbf{P}(X_{n+1} \notin R \mid X_n \in R) = 0$.

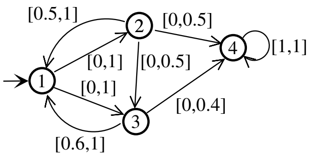


Fig. 4. Interval MC with multiple end components

Definition 7. *Given an Interval MC $\mathcal{I} = (S, s_0, \check{P}, \hat{P})$, a set of states $R \subseteq S$ is an* end component of $\mathcal{I}$ *if*
1) R is strongly connected; 2) $\forall s \in R, t \in S \backslash R. \check{P}_{s,t} = 0$; 3) $\forall s \in R. \sum_{u \in R} \hat{P}_{s,u} \geq 1$.

An end component is maximal if no other end component contains it. In the Interval MC pictured in Fig. 4 we have that $\{1,2\}$ is an end component, $\{1,3\}$ is an end component, and $\{1,2,3\}$ and $\{4\}$ are maximal end components.

Algorithm 1 finds all maximal end components of an Interval MC. It first identifies all candidate end-components and their complement—the obviously transient states; then it propagates transient states backwards to their predecessors who cannot avoid reaching them. The predecessors are pruned from the candidate end-components and the procedure is iterated until a fixed point is reached.

Lemma 8. *Algorithm 1 runs in polynomial time, and upon termination precisely the states that are part of any maximal end component are tagged as* ENDCOMPONENTSTATE*, while the remaining states are tagged as* TRANSIENT.

```
Tag all states of $S$ as UNCHECKED;
Find the strongly connected components (SCCs) of the Interval MC (e.g. with Tarjan's
algorithm) and tag any state not in any SCC as TRANSIENT;
repeat
    foreach SCC $C$ do
        Select a state $s \in C$ tagged UNCHECKED;
        Check that $\forall t \in S \backslash C. \check{P}_{s,t} = 0$. If not, remove $s$ from $C$, tag it TRANSIENT, tag
        UNCHECKED all states in C with a transition to $s$ and select another state;
        Check that $\sum_{u \in C} . \hat{P}_{s,u} \geq 1$. If not, remove $s$ from $C$, tag it TRANSIENT, tag
        UNCHECKED all states in C with a transition to $s$ and select another state;
        Tag $s$ as ENDCOMPONENTSTATE;
    end
until all states in any non-empty SCC are tagged as ENDCOMPONENTSTATE ;
```

Algorithm 1. Find all maximal end components of an Interval MC.

The algorithm to establish finiteness of maximum entropy across all implementations of an Interval MC follows these steps:

Make the Interval MC coherent (see Sect.2);
Find the maximal end components of the Interval MC and call their union S_ω;
If there is a stochastic state in S_ω, then no Maximum Entropy implementation exists.

After finding the maximal end components we check whether each end component state in $\mathcal{I}$ is deterministic. Because the Interval MC is coherent, this check simply amounts to verifying that for each state in each end component there is a successor state with lower bound on transition probability being 1. If this is the case, then there exists a Maximum Entropy implementation for $\mathcal{I}$ with a finite entropy value.

The following theorem states that the above approach to deciding existence of finite maximum entropy implementation is sound and complete:

Theorem 9. *Let $\mathcal{I}$ be an Interval MC and S_ω the union of all its end components. Then $\mathcal{I}$ has no Maximum Entropy implementation iff a state $s \in S_\omega$ is stochastic.*

5.2 Synthesis of a Maximum Entropy Implementation

We have been characterizing the existence of a Maximum Entropy implementation with finite entropy, now we propose a numerical technique to synthesize it with an arbitrary

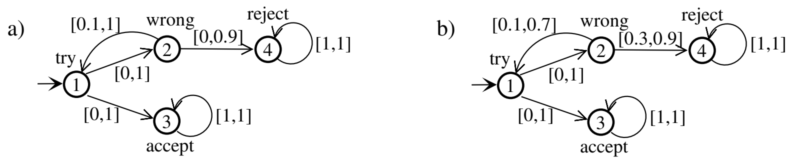


Fig. 5. a) Specification for the Repeated Authentication with unbounded entropy. b) Specification for the Repeated Authentication with bounded entropy.

precision [18]; the Maximum Entropy implementation of the Two-step Authentication in Fig. 2 has been obtained this way. We reduce the problem to solving a multidimensional maximization on convex sets by considering each of the $|S|^2$ transition probabilities $P_{s,t}$ in the chain as different dimensions, each of which can take values in the interval $[\check{P}_{s,t}, \hat{P}_{s,t}]$, generating a convex polytope.

Due to coherence of the Interval MC there exists at least one Markov chain implementing it, so the polytope will be nonempty. We need to add to the system the constraints $\forall s \in S. \sum_{t \in S} P_{s,t} = 1$ to ensure every solution can be interpreted as a MC. Since these constraints are linear, the domain is still a convex polytope. A point in the polytope thus defines a Markov chain. The objective function to maximize is the entropy of such Markov chain, which can be calculated in PTIME as shown in Sect. 4.

This optimization problem for an everywhere differentiable function can be solved using numerical methods. Once the global maximum is found with a numerical algorithm, the parameters $P_{s,t}$ interpreted as a MC give a Maximum Entropy implementation.

Example. Consider the Two-step Authentication in Fig. 1a. The entropy of the system is $H = (-(P_{1,2} \log_2(P_{1,2})) - ((1-P_{1,2}) \log_2(1-P_{1,2})))+P_{1,2}(-(P_{2,4} \log_2(P_{2,4})) - ((1-P_{2,4}) \log_2(1-P_{2,4})))$ under constraints $0 \leq P_{1,2} \leq 1 \wedge 0 \leq P_{2,4} \leq 1$. It is maximal for $P_{1,2} = 2/3$, $P_{2,4} = 0.5$. The Maximum Entropy implementation is shown in Fig. 2.

6 Infinite vs. Unbounded Entropy for Interval MCs

We now give insight about the difference between unbounded and infinite entropy for an Interval MC and give a decision procedure to distinguish the two cases. The infinite case means that it is possible to give non-terminating implementations, while the unbounded case means that all implementations terminate but may leak the whole secret, and thus we cannot give security guarantees for their behavior.

Consider the Repeated Authentication in Fig. 5a; since state 1 can be both recurrent and stochastic but never both, we are in the unbounded case, and in fact it is possible to give implementations with arbitrary entropy. Since the Repeated Authentication is a security scenario, this means that it is possible to give implementations that leak any amount of information about the confidential data, and thus this should be considered an insecure authentication protocol, as it is not possible to give any security guarantee for it.

In this particular case this depends on the fact that we allow the black list to be empty; in this implementation the attacker can try all possible passwords, and thus will eventually leak all of the confidential data. In Figure 5b we show a modified version in which the black list covers at least 30% of the passwords; for this case the Interval MC has a Maximum Entropy implementation, and is thus possible to give a security guarantee.

The idea to discriminate the two cases is to build an implementation that maximizes the end components (in which all states that can be stochastic are stochastic). If this implementation has stochastic states in a strongly connected component, then it will be possible to generate an infinite amount of entropy, otherwise the entropy of any implementation is always finite.

```
Find all maximal end components of the Interval MC;
Modify the transition probabilities so that all end components are closed: for
each end component $\mathcal{R}$, set $\hat{P}_{s,t} = 0$ for all $s \in \mathcal{R}$, $t \notin \mathcal{R}$;
Make the Interval MC coherent again with the coherence algorithm;
If all states in all end components of the coherent Interval MC are deterministic,
then the original Interval MC does not allow infinite entropy implementations;
else it does.
```

After step 2 the Interval MC will still have implementations, since by the definition of end components it's possible to give an implementation that has probability 0 of leaving the end component; we are just forcing it to happen for all our end components and checking if this makes them necessarily deterministic or not.

References

1. de Alfaro, L.: Formal Verification of Probabilistic Systems. Ph.D. thesis, Stanford (1997)
2. Bhargava, M., Palamidessi, C.: Probabilistic Anonymity. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 171–185. Springer, Heidelberg (2005)
3. Biondi, F., Legay, A., Malacaria, P., Wąsowski, A.: Quantifying Information Leakage of Randomized Protocols. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 68–87. Springer, Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-642-35873-9_7`
4. Chatterjee, K., Sen, K., Henzinger, T.A.: Model-Checking ω-Regular Properties of Interval Markov Chains. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 302–317. Springer, Heidelberg (2008)
5. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity Protocols as Noisy Channels. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 281–300. Springer, Heidelberg (2007)
6. Chen, H., Malacaria, P.: Quantifying maximal loss of anonymity in protocols. In: Li, W., Susilo, W., Tupakula, U.K., Safavi-Naini, R. (eds.) ASIACCS. ACM (2009)
7. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security 15 (2007)
8. Cover, T., Thomas, J.: Elements of information theory. Wiley, New York (1991)
9. Girardin, V.: Entropy maximization for markov and semi-markov processes. Methodology and Computing in Applied Probability 6, 109–127 (2004)
10. Jaynes, E.T.: Information Theory and Statistical Mechanics. Physical Review Online Archive (Prola) 106(4), 620–630 (1957)
11. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: LICS, pp. 266–277. IEEE Computer Society (1991)

12. Kozine, I., Utkin, L.V.: Interval-valued finite markov chains. Reliable Computing 8(2), 97–113 (2002)
13. Malacaria, P.: Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow. CoRR abs/1101.3453 (2011)
14. Malacaria, P., Chen, H.: Lagrange multipliers and maximum information leakage in different observational models. In: PLAS 2008, pp. 135–146. ACM, New York (2008)
15. Millen, J.K.: Covert channel capacity. In: IEEE Symposium on Security and Privacy, pp. 60–66 (1987)
16. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley-Interscience (April 1994)
17. Shannon, C.E.: A mathematical theory of communication. The Bell System Technical Journal 27, 379–423 (1948)
18. Stoer, J., Bulirsch, R., Bartels, R., Gautschi, W., Witzgall, C.: Introduction to Numerical Analysis. Texts in Applied Mathematics. Springer (2010)

MAT Learning of Universal Automata

Johanna Björklund[1], Henning Fernau[2], and Anna Kasprzik[2]

[1] Universitet Umeå, Department of Computing Science, S-90750 Umeå, Sweden
johanna@cs.umu.se
[2] Universität Trier, FB IV—Abteilung Informatikwissenschaften
D-54286 Trier, Germany
{fernau,kasprzik}@informatik.uni-trier.de

Abstract. A MAT learning algorithm is presented that infers the universal automaton (UA) for a regular target language, using a polynomial number of queries with respect to that automaton. The UA is one of several canonical characterizations for regular languages. Our learner is based on the concept of an observation table, which seems to be particularly fitting for this computational model, and the necessary notions and definitions are adapted from the literature to the case of UA.

Keywords: universal automata, query learning, observation tables.

1 Introduction

The area of Grammatical Inference (GI) is concerned with algorithms that extrapolate from limited information to infer a formal description of an unknown language. An important concept in this context is the convergence to a certain partition of the target language, which is obtained by splitting and merging sets (or, from the automaton perspective; states). In this paper, we present an algorithm with the objective of inferring the *universal automaton* (UA) for the target language, and in doing so we restrict our attention to automata in which states are non-mergible by definition. We may therefore adopt a general strategy of iteratively dividing states until the conditions for the desired type of description are met. The long-term memory [5] of our learner shall be an *observation table*, which in its most general interpretation fits the characteristics of universal automata more closely than those of any other kind of finite-state automaton. This also means that our way of obtaining an automaton from an observation table is distinctively different from earlier approaches such as [2,3]. Learning of universal automata (in a different learning model) has been successfully applied to some problems from bioinformatics, as reported in [1].

When formalizing a learning task, the information source is of key importance. A substantial amount of work has also been devoted to algorithms that learn by querying an oracle. Angluin [2] introduced the notion of a *minimal adequate teacher* (MAT) to allow for polynomial-time learning of regular languages. This is an oracle capable of answering two types of queries, membership and equivalence queries. Let L be the target language. An *equivalence query* (EQ) is of the form

"Is $\mathcal{A}$ a correct description of L?", and is answered by the oracle either with a simple 'yes', or with a counterexample in the symmetric difference of $\mathcal{L}(\mathcal{A})$ and L (that is, with an element in $c \in (L \setminus \mathcal{L}(\mathcal{A})) \cup (\mathcal{L}(\mathcal{A}) \setminus L)$). *Membership queries* (MQs), on the other hand, are of the type "Is w an element of L?" and are answered with 'yes' or 'no'. In the present article, we adopt the MAT model and require the learner to return the target universal automaton after a finite number of queries.

Whereas [2] focused on learning regular languages by presenting minimal deterministic finite-state automata (DFA) as hypotheses to the teacher, our learner builds *universal automata* (UA), which offer another kind of canonical description for regular languages. A survey of the theory of UA is found in [10].

2 Preliminaries

A *finite-state automaton (FA)* is a tuple $\mathcal{A} = \langle \Sigma, Q, I, F, \delta \rangle$ where Σ is a finite set of *symbols*, Q is the finite set of *states*, $I \subseteq Q$ is the set of *start* or *initial* states, $F \subseteq Q$ is the set of *accepting states*, and $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation.* From the transition relation δ, we derive the functions $\delta^+ : Q \times \Sigma^* \longrightarrow 2^Q$ and $\delta^* : \Sigma^* \longrightarrow 2^Q$. Intuitively, $\delta^+(q, w)$ is the set of all states that can be reached from q on input $w \in \Sigma^*$, and $\delta^*(w)$ is the set of all states that can be reached from an initial state on w. With every state q, we shall associate two sets of strings, $\mathcal{P}_q$ and $\mathcal{F}_q$, the *past* of q and the *future* of q, i.e., for every state $q \in Q$, let $\mathcal{P}_q := \{s \in \Sigma^* \mid q \in \delta^*(s)\}$ and $\mathcal{F}_q := \{e \in \Sigma^* \mid \delta^+(q, e) \cap F \neq \emptyset\}$. A state q is *reachable* if $\mathcal{P}_q \neq \emptyset$ and *co-reachable* if $\mathcal{F}_q \neq \emptyset$. An automaton is *trim* if all of its states are reachable and co-reachable. By keeping only the states that are reachable and co-reachable we obtain the *trimmed version* of an automaton; this can be done in polynomial time and does not change the accepted language.

We identify the automaton $\mathcal{A}$ with the membership predicate for the language that it recognizes. Given $w \in \Sigma^*$, we thus write $\mathcal{A}(w) = 1$ if $\delta^*(w) \cap F \neq \emptyset$. The language accepted by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) := \{w \in \Sigma^* \mid \mathcal{A}(w) = 1\}$. A string language is *regular* if it is accepted by an FA.

An FA $\mathcal{A}$ is *total* if, for every $a \in \Sigma$ and $q \in Q$, there is some $\langle q, a, q' \rangle \in \delta$. Furthermore, $\mathcal{A}$ is a *deterministic FA* (abbreviated DFA) if $\langle q, a, q' \rangle, \langle q, a, q'' \rangle \in \delta$ implies $q' = q''$, otherwise *non-deterministic* (an NFA). For DFA, we may abbreviate $\delta^*(s) = \{q\}$ to $\delta^*(s) = q$, and $\delta^+(s, e) = \{q\}$ to $\delta^+(s, e) = q$ without risk of confusion. We also write $L(w) = 1$ if $w \in L$ for $w \in \Sigma^*$ and $L \subseteq \Sigma^*$, and $L(w) = 0$ if $w \notin L$.

Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language. A pair $\langle X, Y \rangle$ with $X, Y \subseteq \Sigma^*$ is a *subfactor* of L if $X \cdot Y \subseteq L$. A subfactor $\langle X, Y \rangle$ is a *factor* of L if for every $X' \supseteq X$ and $Y' \supseteq Y$, $L \supseteq X'Y'$ implies $X' = X$ and $Y' = Y$. The set $\mathit{fac}(L)$ is the set of all factors of L.

As shown by [10], a language L is regular if and only if $\mathit{fac}(L)$ is finite. The set $Q = \mathit{fac}(L)$ can be viewed as the states of an FA $\mathcal{U}_L = \langle \Sigma, Q, I, F, \delta \rangle$ with

- $I = \{\langle X, Y \rangle \in \mathit{fac}(L) \mid \varepsilon \in X\}$, $F = \{\langle X, Y \rangle \in \mathit{fac}(L) \mid \varepsilon \in Y\}$,
- $\langle \langle X, Y \rangle, a, \langle X', Y' \rangle \rangle \in \delta$ if and only if $XaY' \subseteq L$.

This (unique!) automaton is called the *universal automaton* of L.

For $\langle X, Y\rangle \in \mathit{fac}(L)$, the set X determines Y and vice versa via, for example, $Y = \bigcap_{x\in X} x^{-1}L$. The bijection has several interesting implications [10], e.g.:

$$\langle\langle X, Y\rangle, a, \langle X', Y'\rangle\rangle \in \delta \iff Xa \subseteq X' \iff aY' \subseteq Y \ .$$

Let $L \subseteq \Sigma^*$ be the target language. A triple $T = \langle S, E, \mathit{obs}\rangle$ consisting of two non-empty finite sets $S, E \subseteq \Sigma^*$ and a function $\mathit{obs} : S \times E \longrightarrow \{0, 1\}$ is an *observation table* for L if S is *prefix-closed*, E is *suffix-closed*, obs is a total function with

$$\mathit{obs}(s, e) = \begin{cases} 1 & \text{if } se \in L \text{ is confirmed,} \\ 0 & \text{if } se \notin L \text{ is confirmed.} \end{cases}$$

3 Tables of Subsets

The following ideas, which only require some basic set theory, are fundamental for our approach. Similar notions have been developed in [6,8].

We consider a *universe* $U \times V$ and a *target* $T \subseteq U \times V$. By letting $\mathfrak{U} = 2^U$ and $\mathfrak{V} = 2^V$, we create a *frame* $\mathfrak{U}\times\mathfrak{V}$. An element $(X, Y) \in \mathfrak{U}\times\mathfrak{V}$ is a *subfactor* of T if $X \times Y \subseteq T$. A subfactor (X, Y) of T is a *factor* if, for every subfactor (X', Y') of T, $X \subseteq X'$ and $Y \subseteq Y'$ imply that $X = X'$ and $Y = Y'$. A set $\mathfrak{C} \subseteq \mathfrak{U} \times \mathfrak{V}$ is a *cover with respect to* T if, for every $(x, y) \in T$, there is some $(X, Y) \in \mathfrak{C}$ with $x \in X$ and $y \in Y$. A cover $\mathfrak{C} \subseteq \mathfrak{U} \times \mathfrak{V}$ is a *subfactor cover* (or a *factor cover*) if each $(X, Y) \in \mathfrak{C}$ is a subfactor (or a factor, respectively).

The reasoning behind these definitions is as follows: Consider an alphabet Σ, take $U = \Sigma^*$, $V = \Sigma^*$, and let L be the target language of some learning process. The language L then defines an infinite target table T_L given by $(u, v) \in T_L$ iff $uv \in L$. The condition $X \times Y \subseteq T_L$ is now clearly equivalent to $X \cdot Y \subseteq L$. In the formal language terminology introduced in Section 2, $\langle X, Y\rangle$ is a (sub)factor of L iff (X, Y) is a (sub)factor of T_L, while a (sub)factor cover corresponds to a set of (sub)factors $\{\langle X_i, Y_i\rangle \mid i \in J\}$ of L with $\bigcup_{i\in J} X_i \cdot Y_i = L$.

Lemma 1. *Let $\mathfrak{C} \subseteq \mathfrak{U} \times \mathfrak{V}$ and let $T \subset T' \subseteq U \times V$ be two targets. If $\mathfrak{C}$ is a cover (subfactor cover) with respect to T, then there is some cover (subfactor cover) $\mathfrak{C}' \subseteq \mathfrak{U} \times \mathfrak{V}$ with respect to T', extending $\mathfrak{C}$ in the sense that $\mathfrak{C} \subseteq \mathfrak{C}'$.*

It is tempting to claim the same for factor covers, but unfortunately it does not hold. This is witnessed by $U = \{1\}$, $V = \{a, b\}$, $T = \{(1, a)\}$, and $T' = T \cup \{(1, b)\}$. Here, $\{U \times \{a\}\}$ is a factor cover of T, but $U \times \{a\}$ is not a factor of T', so no cover of T' can both contain $U \times \{a\}$ and be a factor cover of T'. As we shall see, it is possible to obtain a result corresponding to Lemma 1 also for factor covers, but this requires additional notation.

To this end, let us fix a *sub-universe* $S \times E$ of $U \times V$ such that $S \subseteq U$ and $E \subseteq V$. This restriction induces a *sub-frame* $\mathfrak{S} \times \mathfrak{E}$, with $\mathfrak{S} = 2^S \subseteq \mathfrak{U}$ and $\mathfrak{E} = 2^E \subseteq \mathfrak{V}$. Again, let $T \subseteq U \times V$ be our target, and assume that $\mathfrak{C} \subseteq \mathfrak{U} \times \mathfrak{V}$ is a cover with respect to T. The *cover and target induced by* $S \times E$ is then

$\mathfrak{C}|_{S\times E} = \{(X\cap S, Y\cap E) \mid (X,Y)\in\mathfrak{C}\}$, and $T|_{S\times E} = T\cap(S\times E)$, respectively. The names are justified by the elementary Lemma 2.

Lemma 2. *Let $\mathfrak{C}$ be a cover with respect to T. Then $\mathfrak{C}|_{S\times E}$ is a cover with respect to $T|_{S\times E}$. If $\mathfrak{C}$ is a subfactor cover then $\mathfrak{C}|_{S\times E}$ is a subfactor cover.*

Example 3. As we shall see, even if $\mathfrak{C}$ is a factor cover then this need not be the case for $\mathfrak{C}|_{S\times E}$. Let $U\times V$ with $U=\{1,2,3,4\}$ and $V=\{a,b,c\}$ be a universe and $T=\{(1,a),(1,b),(1,c),\ (2,a),(2,b),(3,b),(4,a)\}$ be our target. Let $S\times E$ be a sub-universe with $S=\{1,2\}$ and $E=\{a,b\}$. The target induced by $S\times E$ is $T|_{S\times E}=\{(1,a),(1,b),(2,a),(2,b)\}$. The only factor cover with respect to $T|_{S\times E}$ is $\{S\times E\}$. If we look at the factor cover $\mathfrak{C}=\{\{1\}\times V, \{1,2,4\}\times\{a\}, \{1,2,3\}\times\{b\}\}$ of T then its restriction $\mathfrak{C}|_{S\times E}=\{\{1\}\times E, S\times\{a\}, S\times\{b\}\}$ is not a factor cover of $T|_{S\times E}$. However, by Lemma 2, it is a subfactor cover of $T|_{S\times E}$.

For the reverse direction, where we enlarge rather than restrict the domain, it is possible to embed smaller factor covers into larger ones. We introduce a further notion that becomes important in this context. Let T be a target with the universe $U\times V$. For $X\subseteq U$, $(X, V[X])$ denotes the *right-maximal subfactor induced by* X, i.e., $V[X]$ is the largest subset of V such that $(X,V[X])$ is a subfactor of T, i.e., $X\times V[X]\subseteq T$. For $Y\subseteq V$, $(U[Y],Y)$ analogously denotes the *left-maximal subfactor induced by* Y.

Lemma 4. *For the subset*
- *$X\subseteq U$, $(X,V[X])$ is a subfactor with $V[X]=\{v\in V\mid \forall x\in X:(x,v)\in T\}$.*
- *$Y\subseteq V$, $(U[Y],Y)$ is a subfactor with $U[Y]=\{u\in U\mid \forall y\in Y:(u,y)\in T\}$.*
- *$X\subseteq U$, $(U[V[X]],V[X])$ is a factor, called* the factor induced by X.
- *$Y\subseteq V$, $(U[Y],V[U[Y]])$ is a factor, called* the factor induced by Y.

Let $\mathfrak{C}\subseteq\mathfrak{U}\times\mathfrak{V}$ be a factor cover with respect to the target T.

Lemma 5. *If $\mathfrak{C}$ is a factor cover with respect to $T|_{S\times E}$ then there is a factor cover $\mathfrak{C}'$ with respect to T such that $\mathfrak{C}\subseteq\mathfrak{C}'|_{S\times E}$. This fact is testified by the embedding $f:\mathfrak{C}\to\mathfrak{C}'$, $(X,Y)\mapsto(U[Y],V[U[Y]])$ which satisfies $X\subseteq U[Y]$ and $Y\subseteq V[U[Y]]$.*

Example 6. (cont'd) Starting from the factor cover $\mathfrak{C}'=\{S\times E\}$ of $T|_{S\times E}$, we can use the embedding f in the proof of Lemma 5, which has a fixed-point on $S\times E$, to find the cover $\mathfrak{K}=\mathfrak{C}\uplus\mathfrak{C}'=\{\{1\}\times V, \{1,2,4\}\times\{a\}, \{1,2,3\}\times\{b\}, S\times E\}$ of T. Note that both $\mathfrak{K}$ and $\mathfrak{C}$ are factor covers of T, even though one is a proper subset of the other. This shows that factor covers are not necessarily unique, and that $\mathfrak{C}'\subseteq\mathfrak{K}|_{S\times E}$ may be strict. If we increase T slightly, setting $T'=T\cup\{(2,c)\}$ and using the same restricting set $S\times E$ we would then get $\mathfrak{K}=\{\{1,2,4\}\times\{a\}, \{1,2,3\}\times\{c\}, S\times V\}$. Moreover, $f(S\times E)=S\times V$.

Lemma 7 is important for the analysis of our learning algorithm.

Lemma 7. *The embedding $f:\mathfrak{C}\to\mathfrak{C}'$, $(X,Y)\mapsto(U[Y],V[U[Y]])$ given in Lemma 5 is injective and satisfies $X=U[Y]\cap S$ and $Y=V[U[Y]]\cap E$.*

Proof. Assume that there are (X, Y) and (X', Y') such that $(U[Y], V[U[Y]]) = (U[Y'], V[U[Y']])$. The mapping f is defined in two steps, by first computing the first component from Y and then computing the second component from $U[Y]$. Let us treat the first of these steps. By Lemma 5, X is extended towards $U[Y]$ satisfying $U[Y] \times Y \subseteq T$. Observe that, since (X, Y) was a factor of $T|_{S\times E}$,

$$(U[Y] \setminus X) \cap S = \emptyset \ . \tag{1}$$

Analogously, $(U[Y'] \setminus X') \cap S = \emptyset$. Clearly, $(U[Y], V[U[Y]]) = (U[Y'], V[U[Y']])$ implies that $U[Y] = U[Y']$ and hence that $U[Y] \cap S = U[Y'] \cap S$, which implies $X = X'$ due to Equation 1. A similar argument applies for the second step, yielding $Y = Y'$. This proves the claim.

Lemma 8. *If (X_i, Y_i), $i \in \{1, \ldots, r\}$, are factors of T over $U \times V$ then so are $(\bigcap_{i=1}^r X_i, V[\bigcap_{i=1}^r X_i])$ with $\bigcup_{i=1}^r Y_i \subseteq V[\bigcap_{i=1}^r X_i]$ and $(U[\bigcap_{i=1}^r Y_i], \bigcap_{i=1}^r Y_i)$ with $\bigcup_{i=1}^r X_i \subseteq U[\bigcap_{i=1}^r Y_i]$, if the intersections are not empty.*

In the following sections, T will be alternatively interpreted as the (learning) target and as the observation table of a learning process.

4 Properties of Hypotheses

In this section, we establish the connection between observation tables and universal automata. We begin by defining the *factors of an observation table*, to be contrasted with the previously defined factors of a language. Let $T = \langle S, E, obs\rangle$ be an observation table. A *subfactor* of T is a pair $\langle X, Y\rangle$ with $X \subseteq S$ and $Y \subseteq E$ such that for all $s \in X$ and all $e \in Y$ we have $obs(s, e) = 1$. Analogously, a *factor* of T is a subfactor $\langle X, Y\rangle$ of T such that for every subfactor $\langle X', Y'\rangle$ of T with $X \subseteq X'$ and $Y \subseteq Y'$, we have $\langle X, Y\rangle = \langle X', Y'\rangle$. The set of all factors of T is denoted by $fac(T)$.

Note that in contrast to the classical representation of observation tables introduced above, there is a more general interpretation – clearly, any observation table corresponds to some subset $T \subseteq S \times E$, and the reader can easily verify that the according notions of (sub)factors as discussed in Section 3 coincide.

To differentiate between the notion of factors based on Cartesian products discussed in Section 3 and the one based on the concatenation product, we use parentheses $(,)$ in the first case and pointed brackets $\langle,\rangle$ in the second one.

Let T be an observation table for a language L. The *associated automaton* derived from T is $\mathcal{A}_T = \langle \Sigma, \overline{Q_T}, \overline{I_T}, \overline{F_T}, \overline{\delta_T}\rangle$ with $\overline{Q_T} = fac(T)$, $\overline{I_T} = \{\langle X, Y\rangle \in \overline{Q_T} \mid \varepsilon \in X\}$, $\overline{F_T} = \{\langle X, Y\rangle \in \overline{Q_T} \mid \varepsilon \in Y\}$, and for every $a \in \Sigma$ and $\langle X, Y\rangle, \langle X', Y'\rangle \in \overline{Q_T}$, we have $\langle\langle X, Y\rangle, a, \langle X', Y'\rangle\rangle \in \overline{\delta_T}$ if and only if $X \cdot \{a\} \cdot Y' \subseteq L$. From $\mathcal{A}_T$, we obtain the *associated hypothesis* $\mathcal{H}_T = \langle \Sigma, Q_T, I_T, F_T, \delta_T\rangle$ as the trimmed version of $\mathcal{A}_T$.

In the following, $\mathcal{H}_T$ will be the hypothesis that our learner presents to the teacher, while the condition $X \cdot \{a\} \cdot Y' \subseteq L$ in the construction of $\mathcal{A}_T$ is checked via membership queries to the teacher. Since we allow such queries during the synthesis process, we are guaranteed to find $\mathcal{A}_T$ for *any* observation table T. This sets our algorithm apart from previous MAT learners which require additional properties in their observation tables before synthesizing an automaton. There are examples of observation table T where $\mathcal{A}_T \neq \mathcal{H}_T$.

Remark 9. For $a \in \Sigma$ we define $T \circ a := \langle S, E, obs_a \rangle$ such that $obs_a(s,e) = 1$ if $sae \in L$ is confirmed, and 0 if $sae \notin L$ is confirmed. The definition of $\mathcal{A}_T$ necessitates these auxiliary tables, and an efficient implementation should save the information thus gathered to economize with membership queries. This may speed up the computation but has no bearing on the correctness of the algorithm.

Definition 10. *$\mathcal{A}_T$ is* strongly reachable *if, for all $a \in \Sigma$ and $\langle X', Y' \rangle \in \overline{Q_T}$ and all $xa \in X'$, there is some $\langle X, Y \rangle \in \overline{Q_T}$ with $\langle \langle X, Y \rangle, a, \langle X', Y' \rangle \rangle \in \overline{\delta_T}$ and $x \in X$. Analogously, we can define strong co-reachability.*

Lemma 11. *If $\mathcal{A}_T$ is strongly reachable then it is trim, i.e., $\mathcal{A}_T = \mathcal{H}_T$.*

Definition 12. *An observation table $T = \langle S, E, obs \rangle$ for a language $L \subseteq \Sigma^*$ is* stable *if for every $s, s' \in S$ such that there is $ae \in \Sigma E$ with $L(sae) > L(s'ae)$, there is $e' \in E$ such that $L(se') > L(s'e')$; and for every $e, e' \in E$ such that there is $sa \in S\Sigma$ with $L(sae) > L(sae')$, there is $s' \in S$ such that $L(s'e) > L(s'e')$.*

This is similar to Angluin's consistency condition in her LSTAR algorithm. The proof of the following assertion is pretty straightforward.

Lemma 13. *If T is stable, $\mathcal{A}_T$ is strongly reachable and strongly co-reachable.*

We propose the procedure `MakeStable`(T): Look for $s, s' \in S$ such that there is $ae \in \Sigma E$ with $L(sae) > L(s'ae)$ but there is no $e' \in E$ with $L(se') > L(s'e')$. Add ae to E and fill the table with MQs. Similarly, strings can be added to S.

Lemma 14. *Every time we add an element from $S \cdot \Sigma$ to S, or from $\Sigma \cdot E$ to E in order to make T stable, the number of factors in T increases.*

Proof. Sketch. Suppose that e' is added to E due to Condition 1 in Definition 12. Every factor $\langle X, Y \rangle$ of T with $s \in X$ also had $s' \in X$ since no element in Y can prevent it. By adding e' to E, $\langle X, Y \rangle$ splits into $\langle X, Y \rangle$ and $\langle X', S[X'] \rangle$ with $X' := \{s \in X \mid se' \in L\}$. A similar argument holds when S is enlarged.

As we will make sure that any hypothesis automaton our learner conjectures is from a stable observation table T, we assume stability for all tables in the remainder of this section. This also implies that $fac(T)$ is the state set of any automaton $\mathcal{A}_T$ we consider, as $\mathcal{A}_T = \mathcal{H}_T$.

Lemma 15. *Let $\langle X, Y \rangle \in Q_T$. Then $X = \mathcal{P}_{\langle X,Y \rangle} \cap S$ and $Y = \mathcal{F}_{\langle X,Y \rangle} \cap E$.*

Proof. We only prove $X = \mathcal{P}_{\langle X,Y \rangle} \cap S$ since the part for the future set of $\langle X, Y \rangle$ follows from a symmetrical argument. Let $\langle X, Y \rangle \in Q_T$. We have to prove the following two assertions: (1) If $w \in X$ then $w \in \mathcal{P}_{\langle X,Y \rangle} \cap S$, and (2) If

$w \in \mathcal{P}_{\langle X,Y\rangle} \cap S$ then $w \in X$. The proof is by induction on the length of w. As the induction base, consider $w = \varepsilon$.
(1) Since $\varepsilon \in X$ implies $\langle X,Y\rangle \in I_T$ we have $\varepsilon \in \mathcal{P}_{\langle X,Y\rangle}$ by definition of $\mathcal{P}_{\langle X,Y\rangle}$.
(2) If $\varepsilon \in \mathcal{P}_{\langle X,Y\rangle} \cap S$ then $\langle X,Y\rangle$ must be an initial state of $\mathcal{A}_T$, as our automata do not have transitions on the empty word. By definition, this means that $\varepsilon \in X$.

Now, assume the claim to hold for all states and for all words w of length up to n. Consider some w with $|w| = n+1$. Hence, $w = ua \in S$ for some $u \in \Sigma^n$ and $a \in \Sigma$. The remainder of the proof is as follows for the respective directions:
(1) Consider $w = ua \in X$. As $\mathcal{A}_T$ is strongly reachable (Lemma 13), there is a table factor $\langle X',Y'\rangle$ with $u \in X'$ and $\langle\langle X',Y'\rangle, a, \langle X,Y\rangle\rangle \in \delta_T$. By the induction hypothesis, $u \in \mathcal{P}_{\langle X',Y'\rangle} \cap S$ since S is prefix-closed. As $\langle\langle X',Y'\rangle, a, \langle X,Y\rangle\rangle \in \delta_T$, $w \in \mathcal{P}_{\langle X,Y\rangle} \cap S$.
(2) Assume $w \in \mathcal{P}_{\langle X,Y\rangle} \cap S$. Let $\langle X',Y'\rangle$ be a state that can be passed when leading $w = ua$ into $\langle X,Y\rangle$, with $\langle\langle X',Y'\rangle, a, \langle X,Y\rangle\rangle \in \delta_T$. By the choice of $\langle X',Y'\rangle$, $u \in \mathcal{P}_{\langle X',Y'\rangle} \cap S$ since S is prefix-closed. By the induction hypothesis, $u \in X'$. By the definition of δ_T, in particular for all $y \in Y$, we find that $obs_a(u,y) = 1$. Since $w = ua$, $obs(w,y) = 1$ for all $y \in Y$. As $\langle X,Y\rangle$ is a table factor, we conclude that $w \in X$.

We now turn our attention to a notion of consistency well-known in Learning Theory but less frequently addressed explicitly in Grammatical Inference:

Definition 16. *$\mathcal{A}$ is* T-consistent *if $\mathcal{A}(se) = obs(s,e)$ for every $\langle s,e\rangle \in S \times E$.*

Lemma 17. *The automaton $\mathcal{A}_T$ is T-consistent.*

Proof. Let $\langle s,e\rangle \in S \times E$ with $obs(s,e) = 1$. There is a factor $\langle X,Y\rangle = \langle S[\{e\}], E[S[\{e\}]]\rangle$ such that $s \in X$ and $e \in Y$. Assuming that T is stable, by Lemma 15, $X = \mathcal{P}_{\langle X,Y\rangle} \cap S$ and $Y = \mathcal{F}_{\langle X,Y\rangle} \cap E$, so $\langle X,Y\rangle \in \delta_T^*(s)$ and $\delta_T^+(\langle X,Y\rangle, s) \subseteq F_T$, and consequently $\mathcal{A}_T(se) = 1$.

For the opposite direction, assume that there is an accepting run of $\mathcal{A}_T$ on se. After having read all of s, $\mathcal{A}_T$ must be in some state $\langle X,Y\rangle$ from which it can continue to an accepting state. We thus know that $s \in \mathcal{P}_{\langle X,Y\rangle} \cap S$ and that $e \in \mathcal{F}_{\langle X,Y\rangle} \cap E$. By Lemma 15, $s \in X$ and $e \in Y$, and since $\langle X,Y\rangle$ is a factor, this yields $obs(s,e) = 1$.

We may therefor assume that $\mathcal{A}_T$ is T-consistent, which is useful in the upcoming correctness proof for our inference algorithm. Moreover, by Lemma 15:

Lemma 18. *If the states $\langle X,Y\rangle, \langle X_1,Y_1\rangle, \dots, \langle X_r,Y_r\rangle \in Q_T$ are such that the language $\mathcal{F}_{\langle X,Y\rangle}$ fulfils $\mathcal{F}_{\langle X,Y\rangle} \subseteq \bigcup_{i=1}^r \mathcal{F}_{\langle X_i,Y_i\rangle}$ then we have $Y \subseteq \bigcup_{i=1}^r Y_i$.*

Another important property of observation tables is *closedness*. In our framework, this corresponds to the notion of *saturation*.

Definition 19. *An observation table T for the language L is* saturated *if for every pair of table factors $\langle X,Y\rangle, \langle X',Y'\rangle$ with $XaY' \subseteq L$, there is some $x \in X$ such that $xa \in X'$ and there is some $y \in Y'$ such that $ay \in Y$.*

Lemma 20. *Let T be a saturated observation table. For every natural number r and choice of r table factors $\langle X_i, Y_i\rangle$, $i \in \{1, \ldots, r\}$, it holds that*

$$\bigcap_{i=1}^{r} \mathcal{P}_{\langle X_i, Y_i\rangle} \neq \emptyset \text{ if and only if } \bigcap_{i=1}^{r} X_i \neq \emptyset \ .$$

Proof. The "if" direction is immediate from Lemma 15.

Let x be a string of minimal length that is witness to the falsity of the opposite direction of the lemma. We note that x cannot be the empty string because a state $\langle X, Y\rangle$ is in $\delta_T^*(x)$ if and only if it is an initial state, which it is if and only if $\varepsilon \in X$. Let $x = ua$ for some string $u \in \Sigma^*$ and symbol $a \in \Sigma$, and let $\{\langle X_i, Y_i\rangle \mid i \in \{1, \ldots, r\}\} = \delta_T^*(x)$. Moreover, let $\langle W_i, Z_i\rangle$ with $i \in \{1, \ldots, r\}$ be a selection of factors in $\delta_T^*(u)$ such that $\{\langle\langle W_i, Z_i\rangle, a, \langle X_i, Y_i\rangle\rangle \mid i \in \{1, \ldots, r\}\} \subseteq \delta_T$. By Lemma 8 and the minimality of x, the factor $\langle W, Z\rangle = \langle \bigcap_{i=1}^{r} W_i, E[\bigcap_{i=1}^{r} W_i]\rangle$ exists and since W is a subset of every W_i the set $\{\langle\langle W, Z\rangle, a, \langle X_i, Y_i\rangle\rangle \mid i \in \{1, \ldots, r\}\}$ is contained in δ_T. Definition 19 now lets us pick an arbitrary $w \in W$ such that $wa \in X_j$ for some $j \in \{1, \ldots, r\}$, and because of $WaY_i \subseteq L$ for every Y_i, the maximality of the factors and the containment of $wa \in X_j \subseteq S$, we have $wa \in X_i$ for every $i \in \{1, \ldots, r\}$. This contradicts our assumption concerning x.

We propose the following procedure, which we call `MakeSaturated`(T):

```
For $j \leftarrow 0$ to $|Q_T|$ do:
  for every state $\langle X, Y\rangle$ that contains an $x \in X$ of length $j$ as a shortest word:
    for every state $\langle X', Y'\rangle$ and letter $a$ with $xa \notin X'$, add $xa$ to $X'$.
```

Clearly, adding words to X' amounts in extending S. We proceed similarly with conflicts on the second component. We only argue for conflicts in the first component in the following. Observe that, as we add longer and longer words, no state could have been overlooked by this procedure as inductively we guarantee the containment of words of length j in states $\langle X, Y\rangle$ reachable in j steps; moreover, we could visualize its work (on the first component) as proceeding from the initial states (for $j = 0$) further and further down through the automaton, and we will reach all states by the assumed stability of T, see Lemma 13.

Lemma 21. *Let $T = \langle S, E, obs\rangle$ be an observation table for L and let $T' = \langle S', E', obs'\rangle$ be the observation table resulting from* `MakeSaturated`*(T). Provided that the procedure always terminates, T' is a saturated observation table for L.*

Proof. Consider two factors $p = \langle X, Y\rangle$ and $q = \langle X', Y'\rangle$ of T' with $XaY' \subseteq L$. Our procedure `MakeSaturated`(T) guarantees that among the shortest strings in X, there is some x with $xa \in X'$. By a symmetric argument, among the shortest strings in Y, there is some y with $ay \in Y'$.

Termination will be ensured when we present our learner, but we assume it for now and continue to state Theorem 22, the backbone of our learning algorithm.

Theorem 22. *$\mathcal{A}_T$ is the universal automaton for $\mathcal{L}(\mathcal{A}_T)$.*

The proof is by two lemmata. Lemma 23(which follows from Lemma 15) shows that the states of $\mathcal{A}_T$ satisfy the defining property of UA, and Lemma 24 that the states correspond to factors of the language recognized by $\mathcal{A}_T$.

Lemma 23. *For states $\langle X,Y\rangle, \langle X',Y'\rangle \in Q_T$ and the symbol $a \in \Sigma$, the transition $\langle\langle X,Y\rangle, a, \langle X',Y'\rangle \in \delta_T$ if and only if $\mathcal{P}_{\langle X,Y\rangle} \cdot \{a\} \cdot \mathcal{F}_{\langle X',Y'\rangle} \subseteq \mathcal{L}(\mathcal{A}_T)$.*

Lemma 24. *For all $q \in Q_T$, the pair $\langle \mathcal{P}_q, \mathcal{F}_q\rangle$ is a factor of $\mathcal{L}(\mathcal{A}_T)$.*

Proof. To prove the claim, we can assume $T = \langle S, E, obs\rangle$ to be saturated by Lemma 21. In the following, let $q = \langle X, Y\rangle$. Clearly, $\langle \mathcal{P}_q, \mathcal{F}_q\rangle$ is a subfactor of $\mathcal{L}(\mathcal{A}_T)$. If it were not maximal, there would be an $s \notin \mathcal{P}_q$ with $\{s\} \cdot \mathcal{F}_q \subseteq \mathcal{L}(\mathcal{A}_T)$ or an $e \notin \mathcal{F}_q$ with $\mathcal{P}_q \cdot \{e\} \subseteq \mathcal{L}(\mathcal{A}_T)$. By symmetry, it is sufficient to discuss one case, and prove by induction that $\{s\} \cdot \mathcal{F}_q \subseteq \mathcal{L}(\mathcal{A}_T)$ implies $s \in \mathcal{P}_q$.

Assume then that $\{s\} \cdot \mathcal{F}_q \subseteq \mathcal{L}(\mathcal{A}_T)$, from which we obtain $\{s\} \cdot Y \subseteq \mathcal{L}(\mathcal{A}_T)$ by applying Lemma 15. If $s = \varepsilon$ then $\mathcal{F}_q \subseteq \mathcal{L}$, and hence q is an initial state and ε is trivially in $\mathcal{P}_q$. This proves the base case of the induction.

For the inductive step, assume the claim to be true for all s of length up to n and consider some $s = ua$ of length $n+1$. Let $\delta_T^*(s) = \{\langle X_i, Y_i\rangle \mid 1 \le i \le r\}$ for some $r \in \mathbb{N}$. As $\{s\} \cdot \mathcal{F}_q \subseteq \mathcal{L}(\mathcal{A}_T)$, we have $\delta_T(s) \neq \emptyset$ and moreover $\mathcal{F}_q \subseteq \bigcup_{i=1}^r \mathcal{F}_{\langle X_i,Y_i\rangle}$. By Lemma 18 this yields $Y \subseteq \bigcup_{i=1}^r Y_i$. Let us consider certain factors of T in sequence: (i) For every $i \in \{1,\dots,r\}$, let $\langle Z_i, W_i\rangle \in \delta_T^*(u)$ and $\langle\langle Z_i, W_i\rangle, a, \langle X_i, Y_i\rangle\rangle \in \delta_T$. (ii) For every $i \in \{1,\dots,r\}$, let $X_i' = S[Y_i \cap Y] \supseteq X_i' \cup X$ and $Y_i' = Y_i \cap Y$. Since Y_i and Y overlap, this factor exists, and as $Y_i \cap Y \subseteq Y_i$, we have $\langle\langle Z_i, W_i\rangle, a, \langle X_i', Y_i'\rangle\rangle \in \delta_T$. (iii) Let $\langle Z, W\rangle$ fulfil $Z = \bigcap_{i=1}^r Z_i$ and $W = E[\bigcap_{i=1}^r Z_i] \subseteq \bigcup_{i=1}^r W_i$. Since $u \in \bigcap_{i=1}^r \mathcal{P}_{\langle Z_i,W_i\rangle}$, the intersection $Z = \bigcap_{i=1}^r Z_i$ is not empty, because of Lemma 20 and T being saturated. Moreover, $\langle Z, W\rangle$ is a factor by Lemma 8, and $\langle\langle Z, W\rangle, a, \langle X, Y\rangle\rangle \in \delta_T$, as $zay \in L$ for every $z \in Z$ and $y \in Y$. Now, $\{u\} \cdot \mathcal{F}_{\langle Z,W\rangle} \subseteq L(\mathcal{A}_T)$, and thus $u \in \mathcal{P}_{\langle Z,W\rangle}$ by the induction hypothesis. As $\langle\langle Z, W\rangle, a, \langle X, Y\rangle\rangle \in \delta_T$, we get $s \in \mathcal{P}_{\langle X,Y\rangle}$.

5 Our MAT Learner for Universal Automata

We assume that the target alphabet Σ is given to the learner in advance.

Initialization. The learner starts out with an initial table $T_0 = \langle S_0, E_0, obs_0\rangle$, defined by $S_0 = E_0 = \{\varepsilon\}$.

Loop. The table T_i is completed using MQs, and made stable and saturated using `MakeStable` and `MakeSaturated`. The synthesized automaton $\mathcal{A}_{T_i}$ is passed to the teacher through an EQ. If the teacher accepts $\mathcal{A}_{T_i}$ as the universal automaton for the target language L, then the algorithm terminates successfully. Otherwise, the teacher provides a counterexample w_i. In this case, it adds all prefixes of w_i to S_i and all suffixes to E_i, before reentering the loop with the updated table T_{i+1}.

This algorithm satisfies a number of properties which we state in a sequence of lemmata. By Lemma 17 the learner's hypothesis $\mathcal{A}_T$ is always T-consistent.

Lemma 25. *Either $\mathcal{L}(\mathcal{A}_{T_0}) = \emptyset$ or there is some $A \subseteq \Sigma$ with $\mathcal{L}(\mathcal{A}_{T_0}) = A^*$.*

The languages mentioned in Lemma 25 are exactly those acceptable by any UA with at most one state; our algorithm needs one EQ for these.

Lemma 26. *For each $i \geq 0$, if $\mathcal{A}_{T_{i+1}}$ is presented as a hypothesis, then there is an injective embedding $f_i : Q_i \to Q_{i+1}$ with the property that whenever $\langle X, Y\rangle \mapsto \langle X', Y'\rangle$ then $X = X' \cap S_i$ and $Y = Y' \cap E_i$. A similar statement is true for the intermediate automata obtained before calling* `MakeStable` *or* `MakeSaturated`.

Proof. The proof makes use of notation from Section 3. First, observe that $T_i = T_{i+1}|_{S_i \times E_i}$. Clearly, $Q_i = \mathit{fac}(T_i)$ is a factor cover of T_i. Hence, Lemmas 5 and 7 provide an injective embedding into some factor cover of T_{i+1} which is clearly contained in $Q_{i+1} = \mathit{fac}(T_{i+1})$. The claimed properties $X = X' \cap S_i$ and $Y = Y' \cap E_i$ translate from Lemma 7.

Lemma 27. *For each $i \geq 0$, if the automaton $\mathcal{A}_{T_{i+1}}$ is presented as a hypothesis and if the embedding $f_i : Q_i \to Q_{i+1}$ is bijective then $f_i^{-1} : Q_{i+1} \to Q_i$ is an automaton morphism. The induced mapping $d_i : \delta_{T_{i+1}} \to \delta_{T_i}$ is injective.*

Proof. To avoid a special case, observe that since our algorithm always makes progress in the sense of changing its hypothesis between two rounds, no set of states and no set of transitions considered in this lemma can be empty, as the only possibility to obtain the empty language or the language $\{\varepsilon\}$ as a hypothesis would be with $\mathcal{A}_{T_0}$; we refer to Lemma 25. It remains to show that, whenever there is an a-transition from q to p in $\mathcal{A}_{T_{i+1}}$ then there is an a-transition between the corresponding states $f_i^{-1}(q)$ and $f_i^{-1}(p)$. More concretely, we know that $q, p \in \mathit{fac}(T_{i+1})$, i.e., $q = \langle X_q, Y_q\rangle$ and $p = \langle X_p, Y_p\rangle$. Moreover, Lemma 7 explains that $f_i^{-1}(q) = q' = \langle X'_q, Y'_q\rangle$ with $X'_q = X_q \cap S_i$ and $f_i^{-1}(p) = p' = \langle X'_p, Y'_p\rangle$ with $X'_p = X_p \cap E_i$. By definition, $\langle q, a, p\rangle \in \delta_{T_{i+1}}$ if $xay \in L$ for all $x \in X_q$ and all $y \in Y_p$. Hence, we have $xay \in L$ for all $x \in X'_q$ and $y \in Y'_p$, so that $\langle q, a, p\rangle \in \delta_{T_{i+1}}$ implies $\langle q', a, p'\rangle \in \delta_{T_i}$ as claimed. Clearly, $d_i : \langle q, a, p\rangle \mapsto \langle q', a, p'\rangle \in \delta_{T_i}$ is injective since f_i is a bijection.

Let $\mathcal{U}_L$ be the universal automaton for the target language L with state set $Q = \mathit{fac}(L)$. The following assertion can be seen similar to Lemma 26.

Lemma 28. *For each $i \geq 0$, for $\mathcal{A}_{T_{i+1}}$ there is an injective embedding $f_i : Q_i \to Q$ such that whenever $\langle X, Y\rangle \mapsto \langle X', Y'\rangle$ then $X = X' \cap S_i$ and $Y = Y' \cap E_i$.*

Theorem 29. *The algorithm converges to $\mathcal{U}_L$ within* $\max\{1, |\Sigma|n^3\}$ *many equivalence queries, where n is the number of states of the target automaton $\mathcal{U}_L$.*

Proof. Due to Lemma 28, any hypothesis automaton has at most as many states as $\mathcal{U}_L$. Moreover, Lemma 26 shows that $n_i \leq n_{i+1}$, where $n_j = |Q_j|$ is the number of states of the jth hypothesis. This together with Theorem 22 and the fact that universal automata are unique up to renaming of states shows that the learning algorithm will finally yield the target automaton $\mathcal{U}_L$. In the following reasoning, let m_j denote the number of transitions of the jth hypothesis.

Clearly, $m_j \leq |\Sigma| n_j^2$. Since we always have $\mathcal{A}_{T_j} \neq \mathcal{A}_{T_{j+1}}$ due to the received counterexamples, we can observe two kinds of progress: The first kind is when $n_j < n_{j+1}$. Since $n_{j+1} \leq n$, this kind of progress can occur at most n times. The second is when $n_j = n_{j+1}$ but $m_j > m_{j+1}$. This case is due to Lemma 27. Since $m_j \leq |\Sigma| n_j^2 \leq |\Sigma| n^2$, the second kind of progress can occur at most $|\Sigma| n^2$ times. When combined, the two observations yield the claimed rate of convergence.

Remark 30. An argument similar to the proof of Theorem 29 shows that the procedures `MakeStable` and `MakeSaturated` terminate, as they either increase the number of states or add a small number of table entries a fixed number of times.

6 Discussions, Conclusions and Future Research

To discuss similarities and differences to the famous LSTAR learner of [2], further notation is needed. Let $T = \langle S, E, obs \rangle$ be an observation table. For $s \in S$, let $row[s] = \{e \in E \mid obs(s, e) = 1\}$. Abusing the notation of Section 3, $\langle \{s\}, row[s] \rangle$ is the factor induced by $\{s\}$, as $row[s] = E[\{s\}]$ is the right-maximal subfactor of $\{s\}$. For $row[s] \neq \emptyset$, let $\langle S[row[s]], row[s] \rangle$ be the *row factor* of s. Likewise, let $col[e] = \{s \in S \mid obs(s, e) = 1\}$ and $\langle col[e], E[col[e]] \rangle$ be the *column factor* of e. Let $\mathfrak{R}$ and $\mathfrak{C}$ collect all row and column factors, respectively.

Remark 31. If $\langle X, Y \rangle$ is any factor, then $(X, Y) \in \mathfrak{R} \cup \mathfrak{C}$. This also gives an algorithm for computing $fac(T)$: As long as there exists some yet uncovered (s, e) with $obs(s, e) = 1$, compute the row and column factor of s and e, respectively, and add them to the cover.

Remark 32. If LSTAR constructs a hypothesis from T, then the hypothesized automaton has as state set $\mathfrak{R}$. By way of contrast, our algorithm's hypothesis automaton $\mathcal{A}_T$ has state set $fac(T) = \mathfrak{R} \cup \mathfrak{C}$.

Another difference lies in the definition of a transition function for LSTAR. Define $row[s]_a = \{e \in E \mid obs_a(s, e) = 1\}$ for $a \in \Sigma$. We obtain a transition $\langle s, a, s' \rangle$ for $s, s' \in S$ and $a \in \Sigma$ if $row[s]_a = row[s']$. If for all $a \in \Sigma$ and all $s \in S$ there is some $s' \in S$ with $row[s]_a = row[s']$ then we call T *closed* and all states in the resulting automaton are reachable. If for all $a \in \Sigma$ and all $s, s' \in S$ with $row[s] = row[s']$ we have $row[s]_a = row[s']_a$ then we call T *consistent* and the resulting automaton is deterministic. Hence, our way of deriving an automaton from an observation table also differs from those in [2] or [3] for residual finite-state automata (RFSA). However, it is similar to the notion of *distributional learning* developed by [7] for context-free grammars [6].

As indicated in the last sections of [10], we may find that a generalization of our approach towards the learning of subsets of monoids, not only free monoids, is possible. We are only aware of text learning results for algebraic structures, see [13]. We also encourage to further our approach to learning other structures such as trees, matrices of symbols, or tuples of strings. Alternatively, we can look into other learning models, taking universal automata as our target descriptions. For instance, see [9] for an alternative universal automata learner from positive and negative examples relying on the state merging paradigm.

We hope to report on implementations of our algorithm soon, possibly integrated within existing frameworks like LearnLib [11] or Libalf [4].

Acknowledgements. We thank the referees for their helpful comments and suggestions.

References

1. Álvarez, G.I., Victoria, J.H., Bravo, E., García, P.: A Non-Deterministic Grammar Inference Algorithm Applied to the Cleavage Site Prediction Problem in Bioinformatics. In: Sempere, García (eds.) [12], pp. 267–270
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
3. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-style learning of NFA. In: Boutilier, C. (ed.) Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI, pp. 1004–1009 (2009)
4. 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)
5. Case, J., Jain, S., Ong, Y.S., Semukhin, P., Stephan, F.: Automatic Learners with Feedback Queries. In: Löwe, B., Normann, D., Soskov, I., Soskova, A. (eds.) CiE 2011. LNCS, vol. 6735, pp. 31–40. Springer, Heidelberg (2011)
6. Clark, A.: Distributional Learning of Some Context-Free Languages with a Minimally Adequate Teacher. In: Sempere, García (eds.) [12], pp. 24–37
7. Clark, A.: Learning Context-Free Grammars with the Syntactic Concept Lattice. In: Sempere, García (eds.) [12], pp. 38–51
8. Courcelle, B., Niwinski, D., Podelski, A.: A geometrical view of the determinization and minimization of finite-state automata. Mathematical Systems Theory 24(2), 117–146 (1991)
9. García, P., Vázquez de Parga, M., Álvarez, G.I., Ruiz, J.: Universal automata and NFA learning. Theoretical Computer Science 407(1-3), 192–202 (2008)
10. Lombardy, S., Sakarovitch, J.: The universal automaton. In: Grädel, E., Flum, J., Thomas, W. (eds.) Logic and Automata: History and Perspectives, pp. 457–504. Amsterdam University Press (2008)
11. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next Generation LearnLib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011)
12. Sempere, J.M., García, P. (eds.): ICGI 2010. LNCS, vol. 6339. Springer, Heidelberg (2010)
13. Stephan, F., Ventsov, Y.: Learning algebraic structures from text. Theoretical Computer Science 268(2), 221–273 (2001)

A Graph Polynomial Approach to Primitivity[⋆]

Francine Blanchet-Sadri[1], Michelle Bodnar[2], Nathan Fox[3], and Joe Hidakatsu[2]

[1] Department of Computer Science, University of North Carolina, P.O. Box 26170, Greensboro, NC 27402–6170, USA
blanchet@uncg.edu

[2] Department of Mathematics, University of Michigan, 530 Church Street, Ann Arbor, MI 48109–1043, USA
{mibodnar,joefranz}@umich.edu

[3] Department of Mathematics, Rutgers University, 110 Frelinghuysen Rd., Piscataway, NJ 08854–8019, USA
fox@math.rutgers.edu

Abstract. Recently, Tittmann et al. introduced the *subgraph component polynomial* and showed that its power for distinguishing graphs is quite different from the power of other graph polynomials that appear in the literature such as the matching polynomial, the Tutte polynomial, the characteristic polynomial, the chromatic polynomial, etc. The subgraph component polynomial enumerates vertex induced subgraphs in a given undirected graph with respect to the number of components. We show the use of the subgraph component polynomial to count the number of primitive partial words of a given length over an alphabet of a fixed size, which leads to a method for enumerating such partial words.

1 Introduction

Motivated by social and biological networks, Tittmann et al. [11] introduced the *subgraph component polynomial* $Q(G; x, y)$ of an undirected graph G with n vertices as the bivariate generating function which counts the number of connected components in vertex induced subgraphs. More precisely, $Q(G; x, y) = \Sigma_{i=0}^{n}\Sigma_{j=0}^{n} q_{ij}(G)x^i y^j$, where $q_{ij}(G)$ is the number of vertex induced subgraphs of G with exactly i vertices and j connected components. They related the subgraph component polynomial to other graph polynomials that appear in the literature such as the Tutte polynomial, the universal edge elimination polynomial, etc. (see, for instance, [9] for more information on graph polynomials). They showed several remarkable properties of the subgraph component polynomial, among them is their use to compute the so-called "residual connectedness reliability". They also showed that the problem of computing $Q(G; x, y)$ is $\sharp P$-hard, but that it is fixed parameter tractable when restricting to graph classes that have bounded tree-width and to classes of bounded clique-width.

[⋆] This material is based upon work supported by the National Science Foundation under Grant No. DMS–1060775.

Primitivity is a well-studied topic in combinatorics on words (see, for instance, [5]). It is well-known that the number of primitive words of a given length over an alphabet of a fixed size can be calculated using the Möbius function [8,10]. In this paper, we discuss the use of the above subgraph component polynomial to count the number $P_{h,k}(n)$ of primitive partial words with h holes of length n over a k-letter alphabet. Research on primitive partial words was initiated by the first author in [1]. Partial words, also referred to as strings with don't-cares, may have some undefined positions or holes. In [2], formulas for $h = 1$ and $h = 2$ are given in terms of the formula for $h = 0$ and some bounds are provided for $h > 2$, but no exact formulas are given for $h > 2$. Here, we associate a graph $G_{n,\mathcal{P}}$ with any partial word of length n with period set $\mathcal{P}$ as follows: the vertices represent the positions $0, \ldots, n-1$ and the edges are the pairs $\{i, i+mp\}$, where $0 \leq i < i + mp \leq n-1$, $m \in \mathbb{Z}$, and $p \in \mathcal{P}$. It turns out that $P_{h,k}(n)$ can be expressed in terms of the $Q(G_{n,\mathcal{P}}; x, y)$'s.

In Section 2, we answer the question "How many holes can a primitive partial word of length n over a k-letter alphabet contain?". We show that this number can be expressed in terms of the large factors of n. In Section 3, we describe a general method for counting primitive partial words with the subgraph component polynomial, which leads to a method for the enumeration of primitive partial words. In Section 4, we discuss in particular non-primitive partial words of length pq, where p and q are distinct primes. In doing so, we give a framework for understanding partial words that are exactly p-periodic and exactly q-periodic without being 1-periodic (this relates to a variant of Fine and Wilf's periodicity theorem [6]). In Section 5, we further discuss the computation of $N_{h,k}(n) = \binom{n}{h}k^{n-h} - P_{h,k}(n)$, the number of non-primitive partial words with h holes of length n over a k-letter alphabet, using the subgraph component polynomial. Finally in Section 6, we conclude with some remarks.

We end this section by reviewing a few basic concepts on partial words. Let A be a non-empty finite set, or an *alphabet.* We consider a *partial* word w over A as a word over the enlarged alphabet $A_\diamond = A \cup \{\diamond\}$, where the additional character $\diamond$ plays the role of an undefined position or a hole. For $0 \leq i < n$, the character at position i of w is denoted by $w(i)$. If $w(i) \in A$, then i is defined, otherwise i is a hole. A *full* word is a partial word with an empty set of holes. We denote by $w[i..j)$ the *factor* of w that starts at position i and ends at position $j-1$, and by $|w|$ the *length* of w or the number of characters in w.

If w_1 and w_2 are two partial words of equal length, then w_1 is contained in w_2, denoted by $w_1 \subset w_2$, if $w_1(i) = w_2(i)$ for all defined positions i in w_1. The *greatest lower bound* of w_1 and w_2, denoted by $w_1 \wedge w_2$, is the maximal partial word contained in both w_1 and w_2, i.e, $(w_1 \wedge w_2) \subset w_1$ and $(w_1 \wedge w_2) \subset w_2$, and if $w \subset w_1$ and $w \subset w_2$, then $w \subset (w_1 \wedge w_2)$. For example, $ab\diamond b\diamond a \wedge a\diamond aa\diamond\diamond = a\diamond\diamond\diamond\diamond\diamond$.

For a positive integer p, a partial word w has a *period* of p or w is *p-periodic* if for all positions i, j defined in w such that $i \equiv j \bmod p$, we have $w(i) = w(j)$. A partial word has an *exact period* of p or is *exactly p-periodic* if it is p-periodic and p divides its length. A partial word w is *primitive* if there exists no full word v such that $w \subset v^i$ with $i \geq 2$, equivalently, if there is no proper factor p of

$|w|$ such that w is p-periodic. Clearly, if w is primitive and $w \subset w'$, then w' is primitive.

2 Maximizing Number of Holes in Primitive Words

We define the set of *large factors* $LF(n)$ of an integer n as the set of integers m such that $m < n$, $m \mid n$, and for $t \neq m, t \mid n$, we have $m \nmid t$. For example, when $n = 30$ we have $LF(30) = \{6, 10, 15\}$. Clearly, the large factors of n come from dividing n by its prime factors.

Proposition 1. *Given a primitive partial word of length n which contains the maximum number of holes, set $LF(n) = \{f_1, \ldots, f_m\}$. For any non-hole positions i and j, we have $i - j = c_1 f_1 + \cdots + c_m f_m$ for some $c_i \in \mathbb{Z}$. Moreover, the fewest number of non-holes which rule out all of the large factors of n as periods is $|LF(n)| + 1$.*

Proof. To rule out the large factor f_1, we must use two non-holes i_1, i_2 and they must differ by a multiple of f_1. Note that for each i and j, we have $\operatorname{lcm}(f_i, f_j) = n$, so i_1 and i_2 rule out at most one large factor, i.e., f_1. To rule out the next large factor f_2, there must exist a non-hole position i_3 that differs from i_1 or i_2, say i_1, by some multiple of f_2. Since $i_1 - i_2 = c_1 f_1$ and $i_1 - i_3 = c_2 f_2$ for some $c_1, c_2 \in \mathbb{Z}$, we get $i_2 - i_3 = i_2 - i_1 + i_1 - i_3 = -c_1 f_1 + c_2 f_2$. As noted earlier, the addition of i_3 cannot possibly rule out any large factor other than f_2. Continue in this way until all large factors have been ruled out as periods. After ruling out the first large factor with two non-holes, $|LF(n)| - 1$ non-holes are required to rule out the remaining $|LF(n)| - 1$ large factors. This necessitates a total of $|LF(n)| + 1$ non-hole positions to rule out all large factors of n. $\square$

Theorem 2. *The maximum number of holes that a primitive partial word of length n over an arbitrary alphabet of at least two letters can contain, denoted $\tau(n)$, is $\tau(n) = n - |LF(n)| - 1$. Moreover, the maximum number of holes a primitive word can contain can be achieved using a binary alphabet.*

Proof. We begin with a construction over the binary alphabet $\{a, b\}$ which shows that this number of holes can always be achieved. Let the word w be defined as $w(i) = a$ if $i + 1 \in LF(n)$, $w(i) = b$ if $i = n - 1$, and $w(i) = \diamond$ otherwise. There are $|LF(n)|$ a's and one b, leaving room for exactly $n - (|LF(n)| + 1)$ holes as desired. We observe that it is unnecessary to check for incompatibilities in smaller periods because for any factor q which divides an element p of $LF(n)$, an incompatibility in a period of length p implies an incompatibility in a period of length q, so we need only check that all periods given by our large factor set do not occur in w.

Let $p \in LF(n)$. Since $p \mid n$ we have $n = lp$ for some l. There is an a in position $p - 1$ and a b in position $n - 1 = lp - 1$, so p cannot be a period of w because we have two incompatible positions which differ by a multiple of p. We conclude that this construction yields a primitive word, so $\tau(n) \geq n - |LF(n)| - 1$. By Proposition 1, we require our word to have at least $|LF(n)| + 1$ non-holes, so $\tau(n) \leq n - (|LF(n)| + 1)$. Therefore our construction is maximal. $\square$

3 Counting with the Subgraph Component Polynomial

We start with the graphical representation $G_{n,\mathcal{P}}$ of a partial word of length n with period set $\mathcal{P}$ where edges indicate compatibility: $G_{n,\mathcal{P}} = (V, E)$ is a simple, undirected graph where $V = \{0, \ldots, n-1\}$, and $\{i, j\} \in E$ if and only if there exists $p \in \mathcal{P}$ such that $j = i + mp$ for $m \in \mathbb{Z}$. Fig. 1 gives an example.

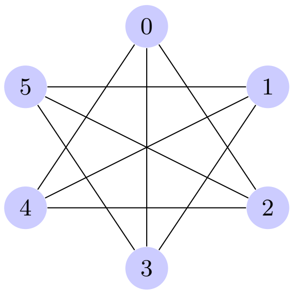


Fig. 1. $G_{n,\mathcal{P}}$ where $n = 6$, $\mathcal{P} = \{2, 3\}$

Let $Q(G; x, y) = \sum_{i=0}^{n} \sum_{j=0}^{n} q_{ij}(G) x^i y^j$ be the *subgraph component polynomial* of a graph G with n vertices, where $q_{ij}(G)$ is the number of induced subgraphs of G with exactly i vertices that have j connected components. For fixed i, we use the notation $Q_i(G; x, y) = \sum_{j=0}^{n} q_{ij}(G) x^i y^j$. For example, the subgraph component polynomial for the graph in Fig. 1 is

$$Q(G_{6,\mathcal{P}}; x, y) = 1 + 6xy + 9x^2y + 6x^2y^2 + 14x^3y + 6x^3y^2 + 15x^4y + 6x^5y + x^6y.$$

The coefficient 9 on x^2y indicates that there are 9 ways to create an induced subgraph of two vertices with a single connected component. This corresponds to the 9 edges in our graph.

Lemma 3. *The number of partial words of length n with h holes over a k-letter alphabet with period set $\mathcal{P}$ is $Q_{n-h}(G_{n,\mathcal{P}}; 1, k)$.*

Proof. The expression $Q_{n-h}(G_{n,\mathcal{P}}; 1, y)$ gives a polynomial in y whose coefficient on y^j is the number of induced subgraphs on $n - h$ vertices that have exactly j connected components. Subgraphs with $n - h$ vertices represent which compatibilities must be satisfied amongst the characters in the word. We associate specific letters with these remaining vertices, noting that each connected component must consist of vertices all associated with the same letter. We have k letter choices to associate with each connected component, so we substitute k for y. There is no double counting between terms because hole placements that lead to different numbers of connected components are necessarily distinct. □

The following theorem gives a formula for the number of non-primitive words of length n with h holes over k letters.

Theorem 4. *If $LF(n) = \{f_1, \ldots, f_m\}$, then*

$$\begin{aligned} N_{h,k}(n) =& \sum_{i=1}^{m} Q_{n-h}(G_{n,\{f_i\}}; 1, k) - \sum_{i \neq j} Q_{n-h}(G_{n,\{f_i,f_j\}}; 1, k) + \cdots \\ &+ (-1)^{l+1} \sum_{i_1 \neq \cdots \neq i_l} Q_{n-h}(G_{n,\{f_{i_1},\ldots,f_{i_l}\}}; 1, k) + \cdots \\ &+ (-1)^{m+1} Q_{n-h}(G_{n,\{f_1,\ldots,f_m\}}; 1, k). \end{aligned}$$

Proof. A word of length n is non-primitive if and only if it has a period equal to one of n's large factors. By Lemma 3, the term in the first sum counts all words with h holes and period f_i. The term in the second sum subtracts those which are double counted because they have periods f_i and f_j. By inclusion-exclusion, the formula counts the total number of words with h holes whose period set contains at least one large factor of n, and thus all non-primitive words. $\square$

To investigate methods of computing the desired subgraph component polynomials, we first recall two important facts from [11, Proposition 15,Theorem 12]: If $G = K_n$ is the complete graph on n vertices, then $Q(K_n; x, y) = y(1+x)^n - y + 1$. While if $G = G_1 \sqcup \cdots \sqcup G_c$ is the disjoint union of c graphs, then $Q(G; x, y) = \prod_{j=1}^{c} Q(G_j; x, y)$.

Proposition 5. *Let $\mathcal{P} = \{p\}$, where p divides n. Then $G_{n,\mathcal{P}}$ is the disjoint union of p graphs isomorphic to $K_{\frac{n}{p}}$ and $Q(G_{n,\mathcal{P}}; x, y) = (y(1+x)^{\frac{n}{p}} - y + 1)^p$.*

Proof. Let V_i consist of the vertex labeled i and all other vertices which differ from i by a multiple of p. This partitions the set of vertices into exactly p equivalence classes, so each induced subgraph G_i is disjoint from every other one. By the definition of $G_{n,\mathcal{P}}$, any two vertices in V_i are connected by an edge so G_i is a complete graph, but no edge connects a vertex in V_i to a vertex in V_j for $i \neq j$. Since there are n/p vertices in V_i, we have G_i isomorphic to $K_{n/p}$ for $0 \leq i < p$. The formula then follows from the abovementioned two facts. $\square$

As an example, we count the number of non-primitive words of length 6 over a 3-letter alphabet with 2 holes. First note that $LF(6) = \{2, 3\}$. By Proposition 5 we have $Q(G_{6,\{2\}}; x, y) = (y(1+x)^3 - y + 1)^2$ and $Q(G_{6,\{3\}}; x, y) = (y(1+x)^2 - y + 1)^3$. Multiplying out the polynomials and looking at terms of interest we have $Q_4(G_{6,\{2\}}; x, y) = 15x^4y^2$ and $Q_4(G_{6,\{3\}}; x, y) = 3x^4y^2 + 12x^4y^3$. Applying Lemma 3, the number of the words that are 2-periodic is $Q_4(G_{6,\{2\}}; 1, 3) = 15(3)^2 = 135$ and the number of the ones that are 3-periodic is $Q_4(G_{6,\{3\}}; 1, 3) = 3(3)^2 + 12(3)^3 = 351$. Recalling the polynomial associated with $G_{6,\{2,3\}}$, the only term containing x^4 is $15x^4y$. Replacing x with 1 and y with 3, the number of words that are both 2- and 3-periodic is $Q_4(G_{6,\{2,3\}}; 1, 3) = 45$. Finally applying Theorem 4, we find that $N_{2,3}(6) = 135 + 351 - 45 = 441$.

The following theorem illustrates how the subgraph component polynomial can arise as a product of smaller polynomials.

Theorem 6. *If $n = c\prod_{i=1}^{d} p_i$ for some $c \in \mathbb{N}$ and primes $p_1, \ldots, p_d$, then*

$$Q(G_{n,\left\{\frac{n}{p_1},\ldots,\frac{n}{p_d}\right\}};x,y) = (Q(G_{\frac{n}{c},\left\{\frac{n}{cp_1},\ldots,\frac{n}{cp_d}\right\}};x,y))^c.$$

Proof. Begin by creating the induced subgraph G_j of $G = G_{n,\left\{\frac{n}{p_1},\ldots,\frac{n}{p_d}\right\}}$ consisting only of vertices labeled i such that $i = j \mod c$ and relabel them 0 through $\frac{n}{c}$. In G, two vertices are connected by an edge if and only if they differ by $\frac{n}{p_1}$, or ..., or $\frac{n}{p_d}$. In G_j, two vertices are connected if and only if they differ by $\frac{n}{cp_1}$, or ..., or $\frac{n}{cp_d}$. Thus, G_j is precisely $G_{\frac{n}{c},\left\{\frac{n}{cp_1},\ldots,\frac{n}{cp_d}\right\}}$. By construction, for $i \neq j$, we have that G_i and G_j are disjoint (with respect to vertices and edges before the relabeling). Viewed before the relabeling, each G_j is equivalent and $G = G_1 \sqcup \cdots \sqcup G_c$, so the result follows from the multiplicativity of the subgraph component polynomial. $\square$

Suppose we wish to compute $N_{2,2}(12)$. By Theorem 6, $Q(G_{12,\{4,6\}};x,y) = (Q(G_{6,\{2,3\}};x,y))^2$. To count words with 2 holes, we must find terms containing x^{10}. Squaring the polynomial computed for $G_{6,\{2,3\}}$, we find that there is precisely one such term: $66x^{10}y^2$. Plugging in $x = 1$ and $y = 2$, we see that there are 264 words of length 12 with 2 holes that have periods 4 and 6. Applying Proposition 5 we have $Q(G_{12,\{4\}};x,y) = (y(1+x)^3 - y + 1)^4$ and $Q(G_{12,\{6\}};x,y) = (y(1+x)^2 - y + 1)^6$, and computing coefficients using the binomial theorem we have $Q_{10}(G_{12,\{4\}};x,y) = 66x^{10}y^4$ and $Q_{10}(G_{12,\{6\}};x,y) = 6x^{10}y^5 + 60x^{10}y^6$. By Lemma 3, there are $66(2)^4 = 1056$ words of length 12 with 2 holes and period 4, and $6(2)^5 + 60(2)^6 = 4032$ with period 6. Finally we apply Theorem 4 to obtain 1056 + 4032 - 264 = 4824 binary non-primitive words of length 12 with 2 holes.

4 Special Cases

We first discuss non-primitive partial words of length p^α for prime number p.

The number of non-primitive partial words with $0 \leq h < p$ holes of prime length p over k letters is $\binom{p}{h}k$ [2]. Summing this over all valid values for h and adding one for the all hole case gives a total of $k2^p - (k-1)$ non-primitive partial words of length p. We extend this result to count the number of non-primitive partial words of length p^α for a positive integer α.

An immediate consequence of the definition of p-periodicity is that a partial word $w = a_0 \cdots a_{n-1}$ of length n is p-periodic if and only if for each integer $0 \leq i < p$, $a_i a_{p+i} \cdots a_{qp+i}$ is 1-periodic, where q is the maximal integer such that $qp + i < n$. We use this fact in the next lemma.

Lemma 7. *Let k, n, and p be positive integers, and write $n = qp + r$ for $0 \leq r < p$. The number of p-periodic partial words of length n over a k-letter alphabet is $\left(k2^{q+1} - (k-1)\right)^r (k2^q - (k-1))^{p-r}$.*

Proof. First, let $p = 1$. Then, $r = 0$ and $n = q$. The fact that the words must be 1-periodic means that there can be at most one specific letter in such words. We can choose that letter in k ways and then each of the $n = q$ positions has two choices: that letter or the $\diamond$ character. Here, we are counting every word exactly once, except for $\diamond^n$, which we are counting k times. Subtracting the extra ones off gives a total of $k2^n - (k-1)$ words of length n that are 1-periodic, which is the desired formula since $n = q$.

Now, let $p > 1$. Split the target words into i smaller words, each of which must be 1-periodic. Exactly r of these words have length $q+1$; the remaining $p-r$ have length q. Also, each of these words can be chosen independently of the others to give a unique word of length n with period p. Hence, by the result for $p = 1$ and the independence property, there are $\left(k2^{q+1} - (k-1)\right)^r (k2^q - (k-1))^{p-r}$ partial words of length n that are p-periodic. □

We can now prove the desired result.

Theorem 8. *Let $\alpha \geq 1$ and p be a prime. There are $(k2^p - (k-1))^{p^{\alpha-1}}$ non-primitive partial words of length p^α over a k-letter alphabet.*

Proof. Any non-primitive partial word of length p^α is $p^{\alpha-1}$-periodic. There are $(k2^p - (k-1))^{p^{\alpha-1}}$ such words by Lemma 7 (set $q := p$, $p := p^{\alpha-1}$, and $r := 0$). □

We now discuss non-primitive partial words of length pq for prime numbers p and q. Since partial words can be exactly p-periodic and exactly q-periodic without being 1-periodic (unlike full words), this case is more challenging than the one of prime power length. We provide the necessary framework for working with these words. We begin with some definitions that help characterize the exceptional words.

Let p and q be positive integers. A partial word w is (p,q)-*special* if it is exactly p-periodic and exactly q-periodic but not $\gcd(p,q)$-periodic. A (p,q)-special partial word w is k-*minimal* if there does not exist a (p,q)-special word using exactly k different letters with fewer holes than w has. (If $k = 2$, we just say that w is *minimal*.) Let $M_k(p,q)$ denote the number of holes in a k-minimal (p,q)-special partial word.

Let X be the set of 1-periodic partial words of length pq such that every character at position $i \bmod p$, for some $0 \leq i < p$, is a hole and every other character is not a hole. We say that a partial word w is $[p,q]$-*special* if it is (p,q)-special or if there exists $v \in X$ such that $w \subset v$. The notion of minimality is the same. Given a $[p,q]$-special partial word w, a *weakening* of w is a partial word obtained by replacing some positions congruent to $i \bmod q$ with holes, for some $0 \leq i < q$. If all non-holes in such indices are replaced, it is a *full weakening*, otherwise it is a *partial weakening*.

Let w_1 and w_2 be partial words, and let $|w_1| = q$ and $|w_2| = p$. Then, we define $\text{str}(w_1, w_2) = w_1^p \wedge w_2^q$. A $[p,q]$-special partial word is *minimal-by-inclusion* if it is not a weakening of $\text{str}(w_1, w_2)$ for any partial words w_1 and w_2.

We now give three important properties of str that are integral to many remaining proofs.

Lemma 9. *Let w_1 and w_2 be full words over the binary alphabet $\{a,b\}$. If p and q are relatively prime, $|w_1| = q$, $|w_2| = p$, and w_2 does not contain all the same* *letter (unless w_1 contains all the same letter, and it is the other letter), then* $\operatorname{str}(w_1, w_2)$ *is $[p, q]$-special, and it is (p, q)-special if each of w_1 and w_2 contains at least one a and at least one b.*

Lemma 10. *Every $[p, q]$-special partial word of length pq over the binary alphabet $\{a, b\}$ is contained in* $\operatorname{str}(w_1, w_2)$ *for some full words w_1 and w_2 such that $|w_1| = q$ and $|w_2| = p$.*

Lemma 11. *A word over the binary alphabet $\{a, b\}$ equals* $\operatorname{str}(w_1, w_2)$ *for some full words w_1 and w_2 such that $|w_1| = q$ and $|w_2| = p$ if and only if it is a minimal-by-inclusion $[p, q]$-special word.*

The next few results relate to counting the number of holes in minimal (or minimal-by-inclusion) special partial words over the binary alphabet $\{a, b\}$. For positive integers p and q, let $h_{p,q}(m, n)$ denote the minimum number of holes possible in $\operatorname{str}(w_1, w_2)$ if $|w_1| = q$, $|w_2| = p$, w_1 and w_2 are full, w_1 contains m b's, and w_2 contains n b's. Also, let $h'_{p,q}(m, n)$ denote the minimum number of holes possible in $\operatorname{str}(w_1, w_2)$ if the above conditions are met and $\operatorname{str}(w_1, w_2)$ is (p, q)-special. (If that condition is impossible to meet, then $h'_{p,q}(m, n) = \infty$.) The following proposition shows that $h_{p,q}(m, n)$ and $h'_{p,q}(m, n)$ can be used interchangeably under some circumstances.

Proposition 12. *If p and q are relatively prime, then $h'_{p,q}(m, n) = h_{p,q}(m, n)$ everywhere that the former is defined, which is for all m and n unless $m = 0$, $n = 0$, $m = q$, or $n = p$.*

Proof. By Lemma 9, $\operatorname{str}(w_1, w_2)$ is automatically (p, q)-special unless $m = 0$, $n = 0$, $m = q$, or $n = p$. □

Here, we show that we can compute h and h' more efficiently than brute-force under some circumstances.

Lemma 13. *If p and q are relatively prime, every possible w_1 and w_2 over the binary alphabet $\{a, b\}$ with $|w_1| = q$, $|w_2| = p$, w_1 and w_2 full, w_1 containing m b's, and w_2 containing n b's results in* $\operatorname{str}(w_1, w_2)$ *containing exactly $h_{p,q}(m, n)$ holes.*

Proof. Let w_1 and w_2 be full words such that $|w_1| = q$, $|w_2| = p$, and all of the a's in both words precede all of the b's. Any other w'_1 and w'_2 with the same values of m, n, p, and q are permutations of w_1 and w_2. If $w_1 = a_0 \cdots a_{q-1}$ and $w_2 = b_0 \cdots b_{p-1}$, consider the following pairing of w_1^p with w_2^q. By the Chinese Remainder Theorem, each index in w_1 is paired exactly once with each index in w_2 (those characters are in the same position as each other in the given powers). Applying any permutation to the letters of w_1 and w_2 does not violate this pairing property. This implies that the resulting greatest lower bound is a permutation of $\operatorname{str}(w_1, w_2)$. In particular, it has the same number of holes. □

The next lemma shows that when we consider any minimal-by-inclusion $[p, q]$-special word, we can consider one of length $\operatorname{lcm}(p, q)$ without loss of generality.

Lemma 14. *If w_1 and w_2 are full words with $|w_1| = q$ and $|w_2| = p$ for some positive integers p and q,* $\operatorname{str}(w_1, w_2)$ *is* $\operatorname{lcm}(p, q)$*-periodic.*

The following theorem is the main result relating to h, and it is used to find minima of h for use in counting non-primitive words.

Theorem 15. *If p and q are relatively prime, then $h_{p,q}(m, n) = mp + nq - 2mn$. And if p and q are positive integers (not necessarily relatively prime), then $h_{p,q}(m, 0) = mp$, $h_{p,q}(0, n) = nq$, and $h_{p,q}(1, 1) = p + q - 2\gcd(p, q)$.*

Proof. Let p and q be relatively prime. By the Chinese Remainder Theorem, when building the greatest lower bound in $\operatorname{str}(w_1, w_2)$ to compute $h_{p,q}(m, n)$ (using any w_1 and w_2 that are allowed, as Lemma 13 allows this) every position in w_1 is paired exactly once with every position in w_2. Hence, every b in w_1 contributes one hole for every a in w_2 and every b in w_2 contributes one hole for every a in w_1. Hence, the total number of holes in $\operatorname{str}(w_1, w_2)$ is $n(q - m) + m(p - n) = mp + nq - 2mn$.

Now, let $m = 1$, $n = 1$, and let p and q be any positive integers. Since each of w_1 and w_2 contain exactly one b, by Lemma 13 we can put the b first, guaranteeing that there is a b in the first $\operatorname{lcm}(p, q)$ characters of $\operatorname{str}(w_1, w_2)$. Since there is only one b per word, there is no more than one b in that block in $\operatorname{str}(w_1, w_2)$. Hence, the b's contribute holes wherever else they come up, which is (in the first $\operatorname{lcm}(p, q)$ positions) a total of $\frac{\operatorname{lcm}(p,q)}{p} + \frac{\operatorname{lcm}(p,q)}{q} - 2$. Then, $h_{p,q}(1, 1)$ is $\left(\frac{\operatorname{lcm}(p,q)}{p} + \frac{\operatorname{lcm}(p,q)}{q} - 2\right)\gcd(p, q) = p + q - 2\gcd(p, q)$. □

The next corollary leads to an easy way of generating (p, q)-special words with the minimal number of holes for some pair (p, q).

Corollary 16. *If p and q are relatively prime, $M_2(p, q) = h_{p,q}(1, 1)$.*

Proof. Note that $M_2(p, q)$ is the minimal value taken by $h'_{p,q}$. We can assume that $0 < m < q$ and $0 < n < p$, as h' is infinite otherwise. We then have $h'_{p,q}(m, n) - h'_{p,q}(1, 1) = mp + nq - 2mn - (p + q - 2) = (m - 1)p + (n - 1)q - 2(mn - 1)$.

Let $f(m, n) = (m - 1)p + (n - 1)q - 2(mn - 1)$. We wish to minimize f over the compact region $[1, q - 1] \times [1, p - 1]$. Examining the partial derivatives, $\frac{\partial f}{\partial m} = p - 2n$ and $\frac{\partial f}{\partial n} = q - 2m$. These partial derivatives imply that the only critical point of f is $(m, n) = \left(\frac{q}{2}, \frac{p}{2}\right)$. We now check the second partial derivatives to determine what type of critical point we have found: $\frac{\partial^2 f}{\partial m^2} = 0$, $\frac{\partial^2 f}{\partial n^2} = 0$, and $\frac{\partial^2 f}{\partial m \partial n} = -2$, so this critical point is a saddle point of f. Hence, the minimum occurs on the boundary, which are the four line segments where $m = 1$ from $n = 1$ to $n = p - 1$, $n = 1$ from $m = 1$ to $m = q - 1$, $m = q - 1$ from $n = 1$ to $n = p - 1$, and $n = p - 1$ from $m = 1$ to $m = q - 1$.

We check the $m = 1$ case; the other three are similar. Here, the function becomes $f(1, n) = (n - 1)q - 2(n - 1)$, which has derivative (with respect to n) $q - 2$, which is not equal to zero unless it is identically zero. Hence, the minimum is an endpoint. The values at the endpoints are $f(1,1) = 0$ and $f(1, p-1) = (p-2)q - 2(p-2) = (p-2)(q-2)$. Therefore, $(1,1)$ is the minimum on this portion of the boundary. Doing the same for $n = 1$ yields the same minimum, and the other two sides yield $(q-1, p-1)$ as another minimum, also with value 0. Therefore, a minimum of f occurs at $(1,1)$, so $h'_{p,q}(1,1) \leq h'_{p,q}(m,n)$. Thus, $M_2(p,q) = h_{p,q}(1,1)$. □

Given a binary p-periodic partial word, a *letter change move of period* p involves choosing an index $0 \leq i < p$ and a letter that does not appear in any of the positions congruent to $i \bmod p$ and replacing all holes in those positions with that letter and all letters in those positions with holes. Note that a letter change move is invertible if it does not yield all holes or begin with all holes; just perform another letter change move on the same index. Call a non-invertible letter change move *degenerate*, and call an invertible one *non-degenerate*. Here, we develop the framework for using letter changes as a counting tool.

Lemma 17. *Let p and q be relatively prime, and let $w = \text{str}(w_1, w_2)$ for some full words w_1 and w_2 with $|w_1| = q$ and $|w_2| = p$ over the binary alphabet $\{a, b\}$. Then, applying any letter change move of period q to w yields $\text{str}(w_3, w_2)$ for some full word w_3 with $|w_3| = q$ and applying any letter change move of period p to w yields $\text{str}(w_1, w_4)$ for some full word w_4 with $|w_4| = p$.*

Proof. We prove the first part. Let $w_1 = a_0 \cdots a_{q-1}$. Let $0 \leq i < q$, and let $w_3 = w_1[0..i)\overline{a_i}w_1[i+1..q)$, where $\overline{a_i}$ indicates the complement of a_i. Let $w = \text{str}(w_1, w_2)$, and let $v = \text{str}(w_3, w_2)$. In every position not congruent to $i \bmod q$, w has the same character as v. In a position congruent to $i \bmod q$, if w has a letter (meaning that w_1^p and w_2^q correspond there), then v has a hole (as w_3^p has a different character there). Similarly, if w has a hole in such a position, then v has a letter there, and it is the same letter that was changed to. Hence, v is precisely the result of applying a letter change of period q to w. □

The next theorem invokes the previous lemma to count minimal-by-inclusion special words.

Theorem 18. *Let p and q be prime numbers. There are $(2^p - 2)\,2^q$ minimal-by-inclusion $[p,q]$-special words of length pq over the binary alphabet.*

Proof. There are 2^q full words of length q and there are $2^p - 2$ full words of length p that are not 1-periodic. Taking str of one word from each of these groups yields precisely the minimal-by-inclusion $[p,q]$-special words of length pq, and each yields a different one. Hence, there are $(2^p - 2)\,2^q$ such words. □

Finally, we use weakenings to count some non-primitive partial words that are not minimal-by-inclusion.

Proposition 19. *Let p and q be prime numbers. For each integer $0 \leq h < q$, there are $\binom{q}{h}(2^p - 2)\,2^{q-h}$ $[p,q]$-special words of length pq over the binary alphabet obtained from exactly h full weakenings of a minimal-by-inclusion $[p,q]$-special word (and there is one such word for $h = q$).*

Proof. The fact that there is one such word for $h = q$ is obvious. Note that the case $h = 0$ is Theorem 18. Now, beginning with a minimal-by-inclusion $[p, q]$-special word w of length pq, let v be a $[p, q]$-special word over the binary alphabet obtained from w by performing h (independent) full weakenings on w for some integer $0 < h < q$. The partial word v has exactly h collections of p holes that are q apart. Each of these hole collections can be identified with a letter. For each hole in the collection, if the letter is the same as the one removed to create that hole, that letter can be replaced there, preserving the specialty property. If it is the other letter and the hole was already a hole, the hole can be replaced by that letter, also preserving the specialty property. Hence, v can be constructed in this way from 2^h different minimal $[p, q]$-special words. Also, by Lemma 11, all $[p, q]$-special words from such weakenings over the binary alphabet for some integer $0 < h < q$ can be obtained this way. Since we are choosing h positions from q to weaken, this gives a total of $\frac{\binom{q}{h}(2^p-2)2^q}{2^h} = \binom{q}{h}(2^p-2)\,2^{q-h}$ $[p, q]$-special words of length pq obtained by applying exactly h full weakenings. □

5 Remarks on Computing $N_{h,k}(n)$

If we know the subgraph component polynomials of graphs from 1 through $n-1$ vertices for any subset of their divisors, how many new subgraph component polynomials must we compute to find $N_{h,k}(n)$ using Theorem 4?

As it turns out, if we compute $N_{h,k}(n)$ starting with $n = 1$ and increasing by 1 each time, we never have to compute more than one new polynomial at each step. In fact, more often than not, we do not need to fully compute anything new. This is a property of Theorem 6. Indeed, by Proposition 5 we know how to compute the first sum from Theorem 4. Now suppose n has large factors $f_1, \dots, f_m$ and we wish to compute the subgraph component polynomial for the graph representation of some proper subset F of $LF(n)$. Then Theorem 6 tells us how to do this in terms of smaller polynomials by taking m to be the product of all $\frac{n}{f_i}$'s such that $f_i \notin F$. Finally, we consider the last term in the formula for $N_{h,k}(n)$. If n is a multiple of the product of its distinct prime factors then we may apply Theorem 6 again. Otherwise, we have a new polynomial to compute. This leads to the following key fact: *All subgraph component polynomials which occur in Theorem 4 arise from polynomials of the form $Q(G_{n,LF(n)}; x, y)$ where n is a product of distinct primes.*

The products $2 \times 3 = 6$, $2 \times 5 = 10$, $2 \times 7 = 14$, $3 \times 5 = 15$, $3 \times 7 = 21$, $2 \times 11 = 22$, and $2 \times 13 = 26$ exhaust all products of two distinct primes less than 30. By computing $Q(G_{n,LF(n)}; x, y)$ for $n = 6, 10, 14, 15, 21, 22, 26$, we can compute $N_{h,k}(n)$ for all $n < 30$ using only powers of these polynomials and Proposition 5. This means computing 7 polynomials as opposed to the 52 that Theorem 4 suggests. More generally, let $\omega(n)$ be the number of distinct prime factors of n. Then the formula for $N_{h,k}(n)$ given by Theorem 4 computes $\sum_{n=1}^{N}(2^{\omega(n)} - 1)$ polynomials to obtain values for all $n < N$. However, simplifications from Theorem 6 reduce this to $D(N)$, the number of products of distinct primes less than N. Using a bound from [7] we have

$$\sum_{n=1}^{N}(2^{\omega(n)} - 1) \geq \sum_{n=1}^{N} \omega(n) \geq N \log(\log N).$$

Moreover, $D(N)$ is bounded from above by N. This implies that if we compute $N_{h,k}(n)$ for all $n < N$ and let N go to infinity, the ratio of what we compute after using Theorem 6 to what is computed using Theorem 4 goes to zero.

6 Conclusion

The results of Section 2 have led to the design and analysis of efficient algorithms for computing all primitively-rooted squares and runs in partial words [3].

The compatibility graphs introduced in Section 3 also appear in the literature under the name *circulant graphs* (see, for instance, [4]). As suggested by one of the referees, a future topic for research would be to investigate special properties of subgraph counting polynomials of circulant graphs to make the computation of $N_{h,k}(n)$, further discussed in Section 5, more efficient.

A WWW server interface at `www.uncg.edu/cmp/research/primitivity3` has been established for automated use of a program that calculates the number of primitive words with a given length, number of holes, and alphabet size.

References

1. Blanchet-Sadri, F.: Primitive partial words. Discrete Applied Mathematics 148, 195–213 (2005)
2. Blanchet-Sadri, F.: Algorithmic Combinatorics on Partial Words. Chapman & Hall/CRC Press, Boca Raton (2008)
3. Blanchet-Sadri, F., Nikkel, J.: Computing primitively-rooted squares and runs in partial words (2012) (preprint)
4. Brown, J.I., Hoshino, R.: On circulants uniquely characterized by their independence polynomials. Ars Combinatoria 104, 363–374 (2012)
5. Dömösi, P., Horváth, S., Ito, M.: Context-Free Languages and Primitive Words. World Scientific (2011) (to appear)
6. Fine, N.J., Wilf, H.S.: Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society 16, 109–114 (1965)
7. Hardy, G.H., Wright, E.M.: An Introduction to the Theory of Numbers. Oxford University Press (2008)
8. Lothaire, M.: Combinatorics on Words. Cambridge University Press, Cambridge (1997)
9. Makowsky, J.A.: From a zoo to a zoology: Towards a general study of graph polynomials. Theory of Computing Systems 43, 542–562 (2008)
10. Petersen, H.: On the language of primitive words. Theoretical Computer Science 161, 141–156 (1996)
11. Tittmann, P., Averbouch, I., Makowsky, J.: The enumeration of vertex induced subgraphs with respect to number of components. European Journal of Combinatorics 32, 954–974 (2010)

Suffix Trees for Partial Words and the Longest Common Compatible Prefix Problem*

Francine Blanchet-Sadri[1] and Justin Lazarow[2]

[1] Department of Computer Science, University of North Carolina, P.O. Box 26170, Greensboro, NC 27402–6170, USA
blanchet@uncg.edu

[2] Department of Mathematics, University of Texas at Austin, 1 University Station C1200 Austin, TX 78712–0233, USA
jlazarow@utexas.edu

Abstract. Suffix trees, introduced by Weiner in 1973, are powerful data structures used to solve many problems on words such as searching for patterns in biological sequences, data compression, text processing, etc. Although they have fallen out of favor in the past years to the more space-efficient suffix arrays, suffix trees are useful for modelling partial words by allowing paths to meet whilst keeping them acyclic through directedness (a partial word may have don't care symbols or holes, which are compatible with any letter of the alphabet over which it is defined). We extend suffix trees to partial words by introducing a suffix directed acyclic graph, with compatibility links, that exhibits all the suffixes while preserving the longest common compatible prefix, lccp, between suffixes. We give an optimal $O(n^2)$ time and space algorithm for constructing the suffix dag of a given partial word w with an arbitrary number of holes of length n over a fixed alphabet by modifying Weiner's algorithm. Our algorithm also computes the lccp array between suffixes of w starting with holes and all other suffixes of w. It possesses the invariant that after the suffix at position i has been processed, the lccp between any suffix starting at or after position i with any suffix starting with a hole can be computed in constant time. As a result, with $O(n^2)$ preprocessing time, finding the lccp of two given suffixes of w requires constant time.

1 Introduction

Given a word $w = w[0..n-1]$ of length n, let \$ be a symbol not appearing in w. The *suffix trie* for w is a tree such that the paths from the root to the n leaves are in one-to-one correspondence with the n suffixes $w[0..n-1]\$, w[1..n-1]\$, \ldots, w[n-1]\$$ of $w\$$, the paths being labelled by these suffixes. Each node is labelled by the label on the path from the root to that node, and a node is called *essential* if it is labelled by one of these n suffixes. It can be constructed by inserting these n suffixes of $w\$$ from the longest to the shortest. If the suffixes

* This material is based upon work supported by the National Science Foundation under Grant No. DMS–1060775.

starting at positions $n-1,\ldots,i$ (or *suffixes at positions* $n-1,\ldots,i$) have been inserted, then the suffix $x = w\$[i-1..n] = w[i-1..n-1]\$$ is inserted as follows. Calculate the *head* of x, which is the longest prefix of x common to the label of a path already present, say $y = w\$[i-1..j-1]$ (the suffix $z = w\$[j..n]$ is called the *tail* of x). Then create a new branch, labelled by z, starting at the node labelled by y. In addition, if a node is labelled by ax, where a is a single character and x is a possibly empty word, there is a *suffix link* from the node for ax to the node for x. The suffix trie of a word of length n can be stored in $O(n^2)$ space [7].

The *suffix tree* for w is the suffix trie for w with edges compressed in a certain way to make the size of the structure linear. The non-essential nodes of degree one are removed (this is called the *compaction* of the trie). The edges become labelled by non-empty words and all the non-essential internal nodes have at least two children. The suffix tree has at most $2n$ nodes and can be stored in $O(n)$ space [7].

Weiner [12] introduced the concept of suffix tree in 1973. Its construction, which can be done in $O(n)$ time for a fixed alphabet size, was simplified by McCreight [9] in 1976 and by Ukkonen [11] in 1995. Farach-Colton et al. [4] in 1997 gave an algorithm that is optimal for all alphabet sizes. Suffix trees have become the basis of many algorithms on words [3,5,10].

We denote by $\text{lcp}(i,j)$ the length of the longest common prefix between the suffixes of w starting at positions i and j. Given two suffixes of w, starting at positions i and j respectively, the *longest common prefix problem* or *lcp problem* is to find the longest word which is a prefix of both of these suffixes. It is equivalent to the *lowest common ancestor problem* of the corresponding suffixes in the suffix tree of w. We say that a subword s of w is *branching* if s is the longest common prefix of distinct suffixes of w. Thus, the suffix tree of w stores all its branching substrings. The ability to compute the lowest common ancestor (and thus the longest common prefix) in constant time and $O(n)$ preprocessing time is a magnificent result [6].

Suffix trees are incredibly powerful data structures for solving some of the harder problems on words: searching for patterns in biological sequences, data compression, text processing, etc. Although they have fallen out of favor in the past years to suffix arrays [8], which are data structures more space-efficient than suffix trees, suffix trees are useful for modelling partial words by allowing paths to meet whilst keeping them acyclic through directedness. We also refer the reader to the directed acyclic word graph (dawg), another data structure with linear time construction that is more space-efficient than a suffix tree [2].

A *partial word* is a sequence that may have undefined positions, called holes and denoted by $\diamond$'s, that match any letter of the alphabet A over which the word is defined (a *full word* is a partial word without holes). We also say that $\diamond$ is *compatible* with each $a \in A$. The compatibility relation can be extended to partial words w and w' of equal length as follows: w is compatible with w', denoted by $w \uparrow w'$, if $w[i] = w'[i]$ whenever $w[i], w'[i] \in A$. The lack of transitivity through compatibility of words is seen by $a \uparrow \diamond$ and $\diamond \uparrow b$, but $a \not\uparrow b$ for distinct letters a and b (for more information on partial words, see [1]). In the context of partial words, we denote by $\text{lccp}(i, j)$ the length of the longest common compatible prefix between the suffixes starting at positions i and j. Given two suffixes of a partial word w, starting at positions i and j respectively, the *longest common compatible prefix problem* or *lccp problem* is to find the longest word which is a compatible prefix of both of these suffixes.

Although there are a great deal of research on algorithms for full words (see, for instance, [3,5,10]), extensions of these algorithms tend to not apply nearly as well for partial words. The abovementioned lack of transitivity through compatibility is one of the more important culprits. In this paper, we extend suffix trees for partial words and show how to use them for computing the longest common compatible prefix between suffixes.

The contents of our paper are as follows: In Section 2, we introduce the suffix directed acyclic graph dag for a partial word w. In Section 3, we give an optimal $O(n^2)$ time and space algorithm for construction of the suffix dag, when w has length n and arbitrarily many holes. We also give such an algorithm when w has length n and a fixed number of holes. In Section 4, we show how to compute the longest common compatible prefix of two given suffixes of w with $O(n^2)$ preprocessing and constant time. In Section 5, we conclude with some remarks.

2 Suffix Directed Acyclic Graphs for Partial Words

We describe a suffix directed acyclic graph (suffix dag) that exhibits all the suffixes of a partial word while preserving the lowest common compatible ancestor or the longest common compatible prefix between suffixes of the word.

We give a description of the basic algorithm that accomplishes this. Let w be a partial word of length n with h holes. Form the suffix trie of w as if the $\diamond$ was simply another letter in the alphabet of which w is over. Then, at every branch in the trie, if any of the branches starts with a $\diamond$, connect the two nodes between the branches that exhibit the maximal path from the parent that is compatible with the other. We call these links, *compatibility links* or *clinks*. Then, to compute the longest common compatible prefix between any two suffixes, we either take one of the nodes given by the lowest compatibility link between nodes in the branches of the suffixes or the lowest common ancestor if no such link exists.

We naturally wish to compress/compact the previously created suffix dag. The rules for compressing nodes is similar to that of the essential nodes for full words. If a node marks the end of a suffix or if the node has degree two without considering compatibility links, we call it essential and cannot compress it. Once we encounter a node with a compatibility link from it, we must stop compression at that node. Thus, the nodes we *can* compress, are the *chains* of nodes of length at least two where the only node allowed to have a compatibility link from it is the last node in the chain.

We now consider the difference between the number of nodes in the suffix tree of a partial word when treating $\diamond$ as another letter in the alphabet and the number of nodes in the compacted suffix dag of the partial word. We must split any chain from the suffix tree in the suffix dag whenever we encounter a node with a compatibility link. We must thus count the number of clinks needed.

Considering the word $a^n \diamond a^n$, we see that the number of compatibility links needed is $n^2 + n$, that is, the nth oblong number. Thus, the number of compatibility links is at least $O(n^2)$ and so, instead of making links at every branch starting with a hole, we will only compute the compatibility links at the root node. We amend our process of computing the longest common compatible prefix:

1. Compute the suffix tree for w treating the $\diamond$ as another letter in the alphabet.
2. Find the branch from the root that starts with a $\diamond$ (this will exist if w has a hole in it); denote it by $\text{branch}_\diamond$.
3. For every subbranch in $\text{branch}_\diamond$ that starts with a $\diamond$ (including the branch itself), add compatibility links between the longest common compatible prefix of these branches with all other suffixes in the tree that are not present in $\text{branch}_\diamond$.
4. Compress the suffix dag.

Fig. 1 illustrates an example of an uncompressed suffix dag created by the process. In this example, $\text{branch}_\diamond$ corresponds to the suffix of $baba\diamond ba$ starting at position 4, that is, $\diamond ba$. There are four clinks, among them is the link between $\diamond b$ and $a\diamond$ since the lccp of $\text{branch}_\diamond$ and the branch corresponding to the suffix starting at position 3, that is, $a\diamond ba$ has length two. Note that there is no clink between $\diamond b$ and ab but there is one between $\diamond ba$ and aba since the lccp of $\text{branch}_\diamond$ and the branch corresponding to the suffix at position 1 has length three.

Lemma 1. *The suffix dag of a partial word with h holes of length n created by the previous process has size in $O(n(h + 1))$ (or more simply $O(nh)$).*

Proof. The suffix tree for w will have a size in $O(n)$. By adding the compatibility links and then compressing, we break a chain into two parts whenever it contains a compatibility link. There are $O(nh)$ possible compatibility links since each corresponds to comparing a suffix starting with a hole to another suffix. Thus, we will add at most $O(nh)$ nodes compared to the suffix tree. $\square$

Let $\mathcal{SD}$ be the compressed suffix dag of a partial word w. If we consider each suffix of w to be labelled as a number (corresponding to start index) in $\mathcal{SD}$, then we define the *lowest common compatible ancestor* or *lcca* between two nodes (suffixes) i and j to be the node of $\mathcal{SD}$ that is a lowest common ancestor of i and j with the possibility of taking the lowest compatibility link between nodes in the branch for suffix starting at position i and the branch for suffix starting at position j if such a link exists. Returning to our example in Fig. 1, for computing $\text{lcca}(3, 4)$ we take the lowest compatibility link, that is, the one between nodes $a\diamond$ and $\diamond b$.

3 Constructing the Suffix Dag

We first provide an optimal algorithm for the construction of the suffix dag for a partial word w of length n with an arbitrary number of holes, that is, the number

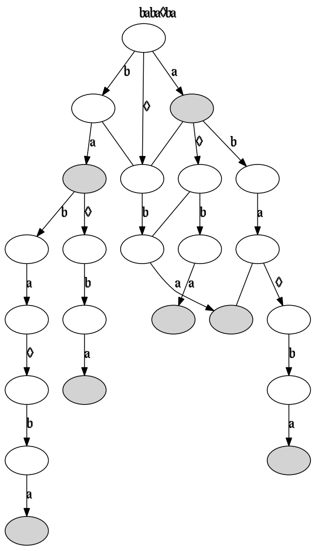


Fig. 1. The uncompressed suffix dag of $baba\diamond ba$ with four compatibility links between nodes b and $\diamond$, $\diamond$ and a, $\diamond b$ and $a\diamond$, and $\diamond ba$ and aba (the essential nodes are colored in grey)

of holes in w is a function of n. Such an algorithm belongs in $O(n^2)$ with regards to both space and running times.

Given a partial word w, denote by $NH(w)[i]$ the position in w of the next hole occurrence at or after position i if it exists. Otherwise, let $NH(w)[i] = -1$ if there are no remaining holes at or after i. For example, if $w = b\diamond\diamond ababa\diamond b$ then $NH(w) = 112888888(-1)$.

Algorithm 1. NH(w)

```
Require: a partial word $w$ of length $n$ and an array $\mathcal{NH}$ of length $n$
Ensure: $NH(w)$ stored in $\mathcal{NH}$ indexed by suffix start position
1: $nextHole \leftarrow -1$
2: for index $pos$ from $n-1$ down to 0 do
3:     if $w[pos] = \diamond$ then
4:         $nextHole \leftarrow pos$
5:     $\mathcal{NH}[pos] \leftarrow nextHole$
6: return $\mathcal{NH}$
```

Lemma 2. *Given a partial word w of length n, Algorithm 1 computes $NH(w)$ in $\Theta(n)$ time.*

We can take advantage of the machinery in Weiner's algorithm for suffix dag construction. We modify Weiner's algorithm, treating the location of appropriate edge/node to break to insert the next suffix as a black box. Whenever a suffix is added, we perform an additional $O(n)$ work to compute the longest common compatible prefix between the new suffix and all suffixes that begin with a hole.

Theorem 3. *Given a partial word w with an arbitrary number of holes of length n over a fixed alphabet, Algorithm 2 runs in $O(n^2)$ time and uses $O(n^2)$ space. It computes both the suffix dag of w and the longest compatible common prefix array between suffixes of w starting with holes and all other suffixes of w. The algorithm possesses the invariant that after we have processed the suffix starting at position i, we can compute the longest common compatible prefix between any suffix starting at or after position i with any suffix starting with a $\diamond$ (anywhere in w) in constant time.*

Proof. Let $w = w[0..n-1]$ be a partial word with an arbitrary number of holes of length n. Let *HoleSuffixes* be the set of positions of w that are holes. Assume we actually compute the suffix tree for every suffix that starts at a position in *HoleSuffixes* first. This does not interfere with any other constructions since no other suffix starts with a $\diamond$. Weiner's algorithm starts with the addition of a node for $w[n-1]\$$ to the tree. Trivially, we know that the length of the longest common compatible prefix between any suffix starting at a member of *HoleSuffixes* and the suffix starting at $n-1$ is 1. Thus, assume we have established this invariant for all suffixes starting at positions $i+1$ through $n-1$ for some integer i. We assume $i+1 \neq 0$.

We now add the suffix starting at position i of w. Weiner's algorithm locates the longest common prefix (lexically) between the suffix of w starting at i and all suffixes of w starting at positions larger than i. Due to the \$ symbol, no suffix is a prefix of another, so either we will introduce a new symbol not yet seen in w, or we will break an edge in the current tree. We denote the longest common prefix value found as $\text{head}(i)$. We define $\text{tail}(i)$ as the word that gives $w[i..n-1] = \text{head}(i)\,\text{tail}(i)$. Weiner's algorithm locates $\text{head}(i)$ in amortized constant time.

Since we assume a fixed alphabet, we allow the lccp values to be computed naively with every member of *HoleSuffixes* if we encounter a suffix that starts with a symbol not previously seen. Thus we add $O(n^2)$ for every letter in the alphabet to the running time of our algorithm.

We assume now that $w[i]$ introduces a symbol previously seen. We denote by v the path label of where we will break the edge to add the new suffix $w[i..n-1]$; we denote d as the length of v. Note that v itself leads to a suffix of w (by not taking any branches) and we denote its starting position as j. We also denote by $CL[s][j]$ the compatibility link between nodes in the branch for suffix starting at j and the branch for suffix starting at a member s of *HoleSuffixes* (we will also refer to this link as the one between j and s). We are then met with the following cases:

Case 4. $CL[s][j]$ is before v.

Algorithm 2. Modified Weiner's Algorithm (Arbitrary Hole Case)

```
Require: a partial word $w = w[0..n-1]$ of length $n$ with $h$ holes, an array $\mathcal{NH}$ of
    length $n$, a list $HoleSuffixes$ of the positions of $w$ with a hole, sorted in ascending
    order, and an empty matrix $\mathcal{LC}$ of size $n \times h$
Ensure: the suffix dag for $w$ and the longest compatible common prefix array between
    suffixes starting with holes and all other suffixes
 1: $\mathcal{NH} \leftarrow \mathbf{NH}(w)$
 2: run Weiner's algorithm on the set of suffixes that start at a position in $HoleSuffixes$

 3: resume Weiner's algorithm on $w$ by adding node for $w[n-1]\$$, that is, suffix tree
    for suffix starting at $n-1$
 4: for $i$ from $n-2$ down to 0 do
 5:    create suffix tree for suffix starting at $i$ from suffix tree starting at $i+1$
 6:    if $w[i]$ is not in $w[i+1..n-1]$ then
 7:       for $s$ in $HoleSuffixes$ do
 8:          compute lccp$(i,s)$ naively, update $\mathcal{LC}[s][i] = $ lccp$(i,s)$
 9:          mark compatibility links between nodes
10:    else
11:       for $s$ in $HoleSuffixes$ do
12:          we are left at the head$(i)$ node which is not the root
13:          let $d$ be the length of the path-label of head$(i)$
14:          let $j$ be the starting position of the suffix that the path from head$(i)$ is a
             prefix of
15:          if compatibility link between $j$ and $s$ is before head$(i)$ then
16:             mark compatibility link and use it for $w[i..n-1]$ to $w[s..n-1]$
17:             $\mathcal{LC}[s][i] = \mathcal{LC}[s][j]$
18:          else
19:             if $w[s+d] = \diamond$ then
20:                $\mathcal{LC}[s][i] = d + \mathcal{LC}[s+d][i+d]$
21:                mark compatibility link
22:             else
23:                $l \leftarrow 0$
24:                for $pos$ from 0 to $\mathcal{NH}[s+d] - (s+d)$ do
25:                   if $pos = \mathcal{NH}[s+d] - (s+d)$ then
26:                      $\mathcal{LC}[s][i] = d + l + \mathcal{LC}[s+d+pos][i+d+pos]$
27:                      mark compatibility link
28:                      break
29:                   if $w[s+d+pos] \not\uparrow w[i+d+pos]$ then
30:                      $\mathcal{LC}[s][i] = d + l$
31:                      mark compatibility link
32:                      break
33:                   $l \leftarrow l+1$
34: traverse to each compatibility link and break edge
```

Algorithm 3. Modified Weiner's Algorithm (Fixed Hole Case)

```
Require: a partial word $w = w[0..n-1]$ of length $n$ with $h(n)$ holes, an array $\mathcal{NH}$ of
    length $n$, a list $\mathit{HoleSuffixes}$ of the positions of $w$ with a hole, sorted in ascending
    order, and an empty matrix $\mathcal{LC}$ of size $n \times h(n)$
Ensure: the suffix dag for $w$ and the longest compatible common prefix array between
    suffixes starting with holes and all other suffixes
 1: $\mathcal{NH} \leftarrow \mathbf{NH}(w)$
 2: run Weiner's algorithm on the set of suffixes at a position in $\mathit{HoleSuffixes}$
 3: compute lccp between the suffix starting at the last hole of $w$ and every suffix
    before it/mark compatibility links
 4: resume Weiner's algorithm on $w$ by adding node for $w[n-1]\$$
 5: for $i$ from $n-2$ down to 0 do
 6:    create suffix tree for suffix at $i$ from suffix tree for suffix at $i+1$
 7:    if $w[i]$ is not in $w[i+1..n-1]$ then
 8:       for $s$ in $\mathit{HoleSuffixes}$ do
 9:          compute lccp$(i,s)$ naively, update $\mathcal{LC}[s][i] = $ lccp$(i,s)$, mark clinks
10:    else
11:       if $w[i] = \diamond$ then
12:          for $j$ from $i-1$ down to 0 do
13:             $\mathit{lexicalLCP} \leftarrow \mathbf{LCP}(i+1, j+1)$
14:             $\mathit{nextHoleJ} \leftarrow \mathcal{NH}[j+1]$ and $\mathit{nextHoleI} \leftarrow \mathcal{NH}[i+1]$
15:             while $-1-i \neq \mathit{nextHoleI} - i = \mathit{nextHoleJ} - j \neq -1-j$ do
16:                $\mathit{nextHoleJ} \leftarrow \mathcal{NH}[\mathit{nextHoleJ}+1]$
17:                $\mathit{nextHoleI} \leftarrow \mathcal{NH}[\mathit{nextHoleI}+1]$
18:             if $\mathit{nextHoleJ} = -1$ then
19:                $\mathit{nextHoleJ} \leftarrow \infty$
20:             else if $\mathit{nextHoleI} = -1$ then
21:                $\mathit{nextHoleI} \leftarrow \infty$
22:             $m \leftarrow \min\{\mathit{nextHoleJ} - j - 1, \mathit{nextHoleI} - i - 1\}$
23:             if $\mathit{lexicalLCP} < m$ then
24:                $\mathcal{LC}[i][j] \leftarrow 1 + \mathit{lexicalLCP}$
25:             else
26:                let $\mathit{holePos}$ be either $\mathit{nextHoleJ}$ or $\mathit{nextHoleI}$, the one that matches
                   $m$, and let $\mathit{otherPos}$ be the other position plus the offset $m$
27:                $\mathcal{LC}[i][j] \leftarrow 1 + m + \mathcal{LC}[\mathit{holePos}][\mathit{otherPos}]$
28:       else
29:          for $s$ in $\mathit{HoleSuffixes}$ do
30:             we are left at the head$(i)$ node which is not the root
31:             let $d$ be the length of the path-label of head$(i)$ and let $j$ be the starting
                position of the suffix that the path from head$(i)$ is a prefix of
32:             if compatibility link between $j$ and $s$ is before head$(i)$ then
33:                mark compatibility link and use it for $w[i..n-1]$ to $w[s..n-1]$
34:                $\mathcal{LC}[s][i] = \mathcal{LC}[s][j]$
35:             else
36:                if $w[s+d] = \diamond$ then
37:                   $\mathcal{LC}[s][i] = d + \mathcal{LC}[s+d][i+d]$ and mark compatibility link
38:                else
39:                   find next hole, update $\mathcal{LC}$ as in Algorithm 2, mark clinks
40: traverse to each compatibility link and break edge
```

Then there is an incompatibility between v and $w[s..n-1]$. The suffix starting at i then satisfies $\text{lccp}(i, s) = \text{lccp}(j, s)$ and so we do not need to do any work. We also copy the compatibility link between j and s for i and s.

Case 5. $CL[s][j]$ is after v and $w[s + d] = \diamond$.

Then $\text{lccp}(i, s) \geq d$. In this case, $\text{tail}(i)$ is a suffix that must start strictly after position i. The invariant of the algorithm then assumes that we can compute $\text{lccp}(i + d, s + d)$ in constant time. Thus, this step takes constant time.

Case 6. $CL[s][j]$ is after v and $w[s + d] \neq \diamond$.

For *pos* from 0 *up until* $NH(w)[s + d] - (s + d)$, where $NH(w)[s + d]$ is the next hole after $s + d$, we must compare $w[i + d + pos]$ with $w[s + d + pos]$ for compatibility. If always compatible, once we reach $NH(w)[s + d] - (s + d)$, we can use the invariant to finish the computation using

$$\text{lccp}(i + d + NH(w)[s + d] - (s + d), s + d + NH(w)[s + d] - (s + d)).$$

This gives a running time for this step in terms of the number of comparisons we must make before reaching the next hole after $s + d$.

Note that the last case takes $O(n)$ time at each suffix insertion. This follows since we need to only compare up until the next hole in the suffix after $w[s + d]$. Thus, each time we invoke that step, we are comparing on a disjoint subset of positions of w, making the sum bounded by n. There are n suffix links to insert and so we will have an $O(n^2)$ algorithm, optimal in the case when h is a function of n itself.

Since we will have $O(n^2)$ compatibility links, we can keep a reference to where we should place them and traverse each link and break the edge accordingly to get the desired suffix dag for w. $\square$

We next give an optimal algorithm, Algorithm 3, for the fixed hole case which requires a bit more machinery and a stronger invariant than the arbitrary hole case. Again, a modified Weiner's algorithm forms the base.

Theorem 4. *Given a partial word w with a fixed number of holes $h(n)$ of length n over a fixed alphabet, Algorithm 3 runs in $O(nh(n))$ time and uses $O(nh(n))$ space. It computes both the suffix dag of w and the longest compatible common prefix array between suffixes of w starting with holes and all other suffixes of w. The algorithm possesses the invariant that after we have processed the suffix starting at position i, we can compute the longest common compatible prefix between any suffix starting at or after position i with any suffix starting with a $\diamond$ (anywhere in w) in constant time.*

Proof. The proof is similar to that of Theorem 3. In Lines 11–27, the algorithm fills the entries of the longest compatible common prefix array between a suffix starting with a hole at position i and the suffixes starting before position i (the suffixes starting after position i have been already looked at since the algorithm processes the suffixes in decreasing order starting from position $n - 1$ down to position 0). For fixed j, $0 \leq j < i$, entry $\mathcal{LC}[i][j]$ is filled in as follows. The algorithm calls $\mathbf{LCP}(i+1, j+1)$, which returns the length $lexicalLCP$ of the longest common prefix between the suffix starting at $i+1$ and the one starting at $j+1$. Note that **LCP** distinguishes $\diamond$ with the letters of the alphabet.

We then look for the positions of the next holes after positions i and j, denoted respectively by $nextHoleI$ and $nextHoleJ$, using $NH(w)$. If $-1-i \neq nextHoleI - i = nextHoleJ - j \neq -1-j$, then $nextHoleI$ and $nextHoleJ$ are at the same distance from i and j respectively. We then update $nextHoleI$ and $nextHoleJ$ with the positions of the next holes after positions $nextHoleI$ and $nextHoleJ$, respectively, and continue until we find two distinct values for $nextHoleI - i$ and $nextHoleJ - j$ with both $nextHoleI$ and $nextHoleJ$ distinct from -1, if any. If $nextHoleI = -1$, then we set $nextHoleI = \infty$, and similarly for $nextHoleJ$. We calculate $m = \min\{nextHoleJ - j - 1, nextHoleI - i - 1\}$.

If $nextHoleI = nextHoleJ = \infty$, then $m = \infty$. Otherwise, we have a hole in either $nextHoleJ$ or $nextHoleI$. We let $holePos$ be either $nextHoleI$ or $nextHoleJ$, the one that gives the minimum m. If the minimum is given by $nextHoleI$, for instance, then $holePos = i+m+1$ and we let $otherPos = j+m+1$. We then compare $lexicalLCP$ with m.

Two cases can happen. If $lexicalLCP < m$, then $\mathcal{LC}[i][j] = 1 + lexicalLCP$ (the 1 comes from position i, being a $\diamond$, is thus compatible with position j). And if $lexicalLCP \geq m$, then $\mathcal{LC}[i][j] = 1 + m + \mathcal{LC}[holePos][otherPos]$. □

To illustrate Lines 11–27 of Algorithm 3, consider $w = b\diamond\diamond\diamond\diamond\diamond\diamond ababa\diamond b^n$ and set $i = 2, j = 1$. We can align the suffixes starting at positions i and j:

i										$nextHoleI$			
2	3	4	5	6	7	8	9	10	11	12	13	14	$\cdots$
$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	a	b	a	b	a	$\diamond$	b	b	$\cdots$
$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	$\diamond$	a	b	a	b	a	$\diamond$	b	$\cdots$
1	2	3	4	5	6	7	8	9	10	11	12	13	$\cdots$
j					$nextHoleJ$								

We can see that $lexicalLCP = \text{lcp}(i+1, j+1) = \text{lcp}(3,2) = 4$, $nextHoleI$ gets computed as 12 and $nextHoleJ$ as 6, and $m = \min\{nextHoleJ - j - 1, nextHoleI - i - 1\} = \min\{6-1-1, 12-2-1\} = 4$. Thus, $holePos = j+m+1 = 1+4+1 = 6$ and $otherPos = i+m+1 = 2+m+1 = 7$. Since $lexicalLCP \geq m$, $\mathcal{LC}[2][1] = 1 + m + \mathcal{LC}[holePos][otherPos] = 1 + 4 + \mathcal{LC}[6][7] = 1 + 4 + 1 = 6$, as desired.

4 Computing the Longest Common Compatible Prefix

The ability to compute the lowest common ancestor (and thus the longest common prefix) in constant time is a magnificent result [6]. We provide a similar result by replacing the $O(n)$ time needed for preprocessing for full words of length n with an $O(n^2)$ time for preprocessing for partial words of length n with $h(n)$ holes. Such preprocessing allows us to obtain a constant time longest common compatible prefix calculation for partial words.

Algorithm 4. LCCP(i, j)

```
Require: a partial word $w = w[0..n-1]$ of length $n$, an empty array $\mathcal{NH}$ of length $n$,
    and non-negative integers $i$ and $j$
Ensure: the length of the longest common compatible prefix of the suffixes of $w$
    starting at positions $i$ and $j$
 1: compute the suffix tree $\mathcal{ST}$ and suffix dag $\mathcal{SD}$ for $w$
 2: if $i \geq n$ or $j \geq n$ then
 3:     return 0
 4: $\mathcal{NH} \leftarrow \mathbf{NH}(w)$
 5: $f_i \leftarrow \mathcal{NH}[i] - i$
 6: $f_j \leftarrow \mathcal{NH}[j] - j$
 7: $l \leftarrow \mathbf{LCP}(i, j)$
 8: if $\mathcal{NH}[i] = \mathcal{NH}[j] = -1$ then
 9:     return $l$
10: if $\mathcal{NH}[i] = -1$ then
11:     $m \leftarrow f_j$
12: else if $\mathcal{NH}[j] = -1$ then
13:     $m \leftarrow f_i$
14: else
15:     $m \leftarrow \min\{f_i, f_j\}$
16: if $m = 0$ then
17:     return $1 + \mathbf{LCCP}(i + 1, j + 1)$
18: else if $l < m$ then
19:     return $l$
20: else
21:     return $m + \mathbf{LCCP}(i + m, j + m)$
```

Theorem 5. *There exists an algorithm that, with $O(n^2)$ preprocessing, can answer the longest common compatible prefix problem between any two positions in an input partial word with an arbitrary number of holes of length n in constant time for a fixed alphabet.*

Proof. Let w be a given partial word of length n. Referring to Algorithm 4, we first build the suffix tree, $\mathcal{ST}$, for w treating the $\diamond$ symbol as just another letter in the alphabet of which w is over. We make use of one of the many algorithms to then process $\mathcal{ST}$ in $O(n)$ time to allow for constant time lcp calculation. If we use **LCP**, we retrieve the longest common prefix in constant time from $\mathcal{ST}$ (with the necessary preprocessing). We also build the suffix dag for w, $\mathcal{SD}$. We use the $\mathcal{LC}$ table computed by Algorithm 2. We also use a copy of $NH(w)$, denoted $\mathcal{NH}$, which is produced by Algorithm 1.

Let i and j be two positions in w. We want to calculate $\text{lccp}(i, j)$ in constant time. Using $\mathcal{NH}$, we can calculate the positions of the first hole in the suffixes of w starting at positions i and j, denoting these respective values by f_i and f_j. We compute $l = \text{lcp}(i, j)$ in $\mathcal{ST}$ in constant time. If $\mathcal{NH}[i] = \mathcal{NH}[j] = -1$, then both positions i and j start suffixes that are full. We can just return l in this case. Otherwise, one of the suffixes has a hole in it. Let m denote the minimum of f_i and f_j that is non-negative; in other words, if $\mathcal{NH}[i] = -1$ or $\mathcal{NH}[j] = -1$, then let m denote f_j or f_i respectively (since we are assuming both are not negative now).

Suppose $m = 0$. Then the suffix starting at position i or the suffix starting at position j starts with $\diamond$, and $\text{lccp}(i, j) = 1 + \text{lccp}(i + 1, j + 1)$. Otherwise, suppose $l < m$. Then $\text{lccp}(i, j) = l$. Finally, suppose $l \geq m$. Then $\text{lccp}(i, j) = m + \text{lccp}(i + m, j + m)$. But since $w[i + m]$ or $w[j + m]$ starts with $\diamond$, we can compute $\text{lccp}(i+m, j+m)$ in constant time. Thus, $\mathbf{LCCP}(i, j)$ runs in $< O(n^2), 1 >$ time. $\square$

For example, if $w = b\diamond\diamond ababa\diamond b$ then Algorithm 4 computes $\text{lccp}(8, 10) = 0$, $\text{lccp}(7, 9) = \text{lcp}(7, 9) = 0$, $\text{lccp}(6, 8) = 1 + \text{lccp}(6 + 1, 8 + 1) = 1$, and $\text{lccp}(3, 5) = \text{lcp}(3, 5) + \text{lccp}(3 + 3, 5 + 3) = 3 + 1 = 4$.

5 Conclusion

Constructing the suffix dag for partial words may find applications other than computing the longest common compatible prefix between suffixes, which was described in our paper. They include: finding the longest compatible repeated substring, finding the longest common compatible substring, etc. Application areas include computational biology, data compression, etc.

References

1. Blanchet-Sadri, F.: Algorithmic Combinatorics on Partial Words. Chapman & Hall/CRC Press, Boca Raton (2008)
2. Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. Theoretical Computer Science 40, 31–55 (1985)
3. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
4. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. Journal of the Association for Computing Machinery 47, 987–1011 (2000)
5. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, Cambridge (1997)
6. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13, 338–355 (1984)
7. Lothaire, M.: Applied Combinatorics on Words. Cambridge University Press, Cambridge (2005)
8. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22, 935–948 (1993)
9. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the Association for Computing Machinery 23, 262–272 (1976)
10. Smyth, W.F.: Computing Patterns in Strings. Pearson Addison-Wesley (2003)
11. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14, 249–260 (1995)
12. Weiner, P.: Linear pattern matching algorithm. In: 14th Annual IEEE Symposium on Switching and Automata Theory, SWAT 1973, pp. 1–11 (1973)

Dynamic Communicating Automata and Branching High-Level MSCs

Benedikt Bollig[1,⋆], Aiswarya Cyriac[1,⋆], Loïc Hélouët[2], Ahmet Kara[3,⋆⋆], and Thomas Schwentick[3,⋆⋆]

[1] LSV, ENS Cachan, CNRS & INRIA, France
[2] INRIA/IRISA Rennes, France
[3] Lehrstuhl Informatik 1, TU Dortmund, Germany

Abstract. We study dynamic communicating automata (DCA), an extension of classical communicating finite-state machines that allows for dynamic creation of processes. The behavior of a DCA can be described as a set of message sequence charts (MSCs). While DCA serve as a model of an implementation, we propose branching high-level MSCs (bHMSCs) on the specification side. Our focus is on the implementability problem: given a bHMSC, can one construct an equivalent DCA? As this problem is undecidable, we introduce the notion of executability, a decidable necessary criterion for implementability. We show that executability of bHMSCs is EXPTIME-complete. We then identify a class of bHMSCs for which executability effectively implies implementability.

1 Introduction

Communicating automata (CA) [7] are a popular model of boolean concurrent programs, in which a fixed finite number of finite-state processes exchange messages through unbounded FIFO channels. One particular research branch considers a semantics of CA in terms of message sequence charts (MSCs). MSCs propose a visual representation of system executions, can be composed by formalisms like high-level MSCs (HMSCs), and are standardized by the ITU [13]. A natural question in this context is the implementability problem, which asks if a given HMSC can be translated into an equivalent CA [11,1,12,20,10,17,9].

Most previous formal approaches to communicating systems and MSCs restrict to a *fixed finite* set of processes. This limits their applicability, as, nowadays, many applications are designed for an open world, where the participating actors are not entirely known in advance. Example domains include mobile computing and ad-hoc networks. In [4], dynamic communicating automata (DCA) were introduced as a model of programs with process creation. In a DCA, a process may (i) send and receive messages, or (ii) spawn a new process which is equipped with a unique process identifier (pid). Pids can be stored in registers and be exchanged through messages. The use of registers in DCA suggests close

⋆ Supported by DIGITEO LoCoReP and LIA InForMel.
⋆⋆ We acknowledge the financial support by the German DFG, grant SCHW 678/4-1.

connections with register automata (also known as finite-memory automata) and formal languages over infinite alphabets (cf. [21] for an overview).

DCA are inherently hard to analyze and to synthesize. To facilitate the specification of dynamic systems, we introduce branching HMSCs (bHMSCs). Just like DCA generalize CA, bHMSCs extend HMSCs. They are based on branching automata [15,16], which rely on a natural principle of distributed computing: a process can start a number of parallel subprocesses and resume its activity once these subprocesses terminate. Each subprocess may start some subclients so that the number of processes running in parallel is a priori not bounded. Like DCA, bHMSCs use finitely many registers to store pids. In a sense, bHMSCs combine branching automata and register automata.

In this paper, we study the implementability question: given a bHMSC, is there an equivalent DCA? This question is undecidable already in the case of a bounded number of processes [12]. Therefore, we consider the notion of executability, a necessary condition for implementability, which amounts to the question if, in every scenario, communicating processes may know each other at the time of communication. We prove executability of bHMSCs to be EXPTIME-complete. Moreover, we identify the fragment of guarded join-free bHMSCs, for which executability and implementability coincide. In this case we also provide an exponential construction of an equivalent DCA.

Related Work. A first step towards MSCs over an evolving set of processes was made in [14], where MSO model checking is shown decidable for *fork-and-join MSC grammars.* Branching HMSCs are similar to these grammars, but take into account pids as message contents and distinguish messages and process creation. Moreover, (implementable) subclasses can be identified more easily. Nevertheless, several of our results apply to the formalism from [14] once the latter is adjusted to our setting. In [5], an MSC semantics was given for the π-calculus. Note that the problems studied in [14] and [5] are very different from ours and do not distinguish between a specification and an implementation.

The present paper supersedes [4] in several aspects. Branching HMSCs are more expressive than the previous formalism, simpler to understand, and more adequate, since they are based on a natural, well-established extension of finite automata to parallelism. Moreover, we extend DCA in such a way that messages themselves can carry (visible) process identifiers. This aspect is important and frequently used (e.g., in the leader election protocol). Finally, we provide tight complexity bounds for the executability problem and solve the implementability problem for a class of specifications that cannot be handled by [4].

Other formalisms with dynamic process creation (not necessarily involving message passing) can be found, for example, in [8,18,6,2]. However, these papers consider neither an MSC based semantics nor implementability aspects.

Outline. In Section 2, we define MSCs. Branching HMSCs and DCA are presented in Sections 3 and 4, respectively. In Section 5, we study executability. Section 6 identifies a fragment of bHMSCs for which executability and implementability coincide. We conclude in Section 7. Proofs can be found in [3].

2 Dynamic Message Sequence Charts

For sets A and B, let $[A \rightharpoonup B]$ denote the set of partial mappings from A to B. We identify $f \in [A \rightharpoonup B]$ with the set $\{a \mapsto f(a) \mid a \in \mathrm{dom}(f)\}$. A *ranked alphabet* is a nonempty finite set A where every letter $a \in A$ has an arity $\mathit{arity}(a) \in \mathbb{N}$.

Let P be a set of *process names* (or, simply, *processes*). Later, P will be instantiated either by the infinite set $\mathbb{P} = \{0, 1, 2, \ldots\}$ of *process identifiers* (pids, for short), or by a finite set of registers. We fix a ranked alphabet A of *message labels*. The set of *messages* (over P) is defined as $A(P) \overset{\text{def}}{=} \{a(p_1, \ldots, p_n) \mid a \in A$, $n = \mathit{arity}(a)$, and $p_1, \ldots, p_n \in P\}$.

A message sequence chart (MSC) consists of a number of processes. Each process $p \in P$ is represented by a set of events E_p, totally ordered by a direct-successor relation $\lhd_{\mathsf{proc}}$. Every event has a *type* from $\mathcal{T} = \{\mathsf{start}, \mathsf{spawn}, !, ?\}$. The minimal event of a process has type start. Subsequent events can then execute spawn (spawn), send (!), or receive (?) actions. The relation $\lhd_{\mathsf{msg}}$ associates each send event with a unique receive event which is always on a different process. The exchange of messages between two processes has to conform with a FIFO policy. Similarly, $\lhd_{\mathsf{spawn}}$ relates a spawn event $e \in E_p$ with the unique start event of a different process $q \neq p$, meaning that p has created q.

Definition 1 (MSC). *A* message sequence chart (MSC) *over A and P is a tuple $M = (E, \lhd, \lambda, \mu)$ where E is a nonempty finite set of* events*, $\lhd$ is the edge relation, which is partitioned into $\lhd_{\mathsf{proc}} \uplus \lhd_{\mathsf{spawn}} \uplus \lhd_{\mathsf{msg}}$, the mapping $\lambda : E \to \mathcal{T} \times P$ assigns a type and a process to each event, and $\mu : \lhd_{\mathsf{msg}} \to A(P)$ labels a message edge with a message. For each type $\theta \in \mathcal{T}$, we let $E_\theta \overset{def}{=} \{e \in E \mid \lambda(e) \in \{\theta\} \times P\}$. We define the mapping $\mathit{pid} : E \to P$ such that $\mathit{pid}(e) = p$ if $\lambda(e) \in \mathcal{T} \times \{p\}$. Accordingly, for $p \in P$, set $E_p \overset{def}{=} \{e \in E \mid \mathit{pid}(e) = p\}$. We require the following:*

1. *$(E, \lhd^*)$ is a partial order with a unique minimal element $\mathit{init}(M) \in E_{\mathsf{start}}$,*
2. *$\lhd_{\mathsf{proc}} \subseteq \bigcup_{p \in P}(E_p \times E_p)$ and, for each $p \in P$, $\lhd_{\mathsf{proc}} \cap (E_p \times E_p)$ is the direct-successor relation of some total order on E_p,*
3. *$E_{\mathsf{start}} = \{e \in E \mid$ there is no $e' \in E$ such that $e' \lhd_{\mathsf{proc}} e\}$,*
4. *$\lhd_{\mathsf{spawn}}$ and $\lhd_{\mathsf{msg}}$ are subsets of $\bigcup_{p,q \in P \mid p \neq q}(E_p \times E_q)$,*
5. *$\lhd_{\mathsf{spawn}}$ induces a bijection between E_{spawn} and $E_{\mathsf{start}} \setminus \{\mathit{init}(M)\}$,*
6. *$\lhd_{\mathsf{msg}}$ induces a bijection between $E_!$ and $E_?$ satisfying the following (FIFO): for $e_1, e_2 \in E_p$ and $f_1, f_2 \in E_q$ with $e_1 \lhd_{\mathsf{msg}} f_1$ and $e_2 \lhd_{\mathsf{msg}} f_2$, we have $e_1 \lhd^*_{\mathsf{proc}} e_2$ iff $f_1 \lhd^*_{\mathsf{proc}} f_2$.*

The set of MSCs over A and P is denoted by $\mathbb{MSC}(A, P)$.

MSCs enjoy a natural graphical representation. Figure 1 depicts the MSCs $M(n)$ and M_0 over $A = \{a, b, c\}$ and $\mathbb{P}$, where $\mathit{arity}(a) = 1$ and $\mathit{arity}(b) = \mathit{arity}(c) = 0$. The events are the endpoints of arrows. Each arrow is either an element of $\lhd_{\mathsf{spawn}}$ (those with two arrow heads) or an element of $\lhd_{\mathsf{msg}}$ (those with one arrow head and a label from $A(\mathbb{P})$). The relation $\lhd_{\mathsf{proc}}$ orders (top-down) two consecutive points located on the same process line. Event $\mathit{init}(M)$, which is located on the process with pid 0, is depicted as a small circle.

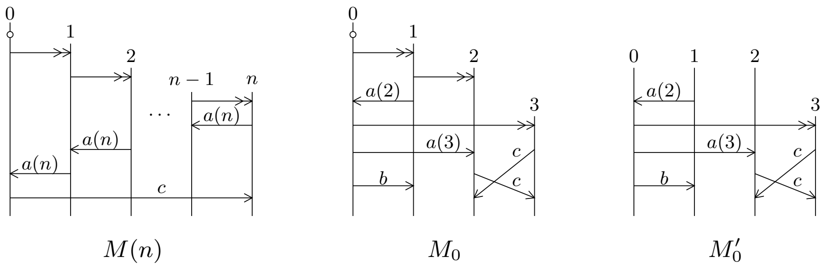
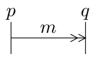
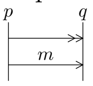


Fig. 1. Two MSCs and a partial MSC

We do not distinguish MSCs that differ only in their event names. We say that two MSCs over A and $\mathbb{P}$ are *equivalent* if one can be obtained from the other by a renaming of pids. The equivalence class of M is denoted $[M]$. Moreover, for a set L of MSCs, we let $[L] = \bigcup_{M \in L}[M]$. We say that L is *closed* if $L = [L]$.

Depending on the application, a spawn in an MSC may have different interpretations, such as create subprocess, contact server, etc. In some cases, one may therefore wish to communicate a message to the new process. This can be simulated in our framework by a message edge that immediately follows a spawn. For a message m, we will actually use 

as an abbreviation for 

.

3 Branching High-Level Message Sequence Charts

In this section, we propose a generalization of HMSCs that is suited to our dynamic setting. It is inspired by branching automata over series-parallel pomsets [15,16]. An MSC can be seen as one single execution of a distributed system. To generate infinite collections of MSCs, specification formalisms usually provide a concatenation operator. It will allow us to append to an MSC a partial MSC, which does not necessarily have start events on each process.

Definition 2 (partial MSC). *Let $M = (E, \lhd, \lambda, \mu) \in \mathbb{MSC}(A, P)$ and let $E' \subseteq E$ be a nonempty upward-closed set containing only* complete *messages and spawning pairs: for all $(e, f) \in \lhd^* \cup \lhd_{\mathsf{msg}}^{-1} \cup \lhd_{\mathsf{spawn}}^{-1}$, we have that $e \in E'$ implies $f \in E'$. Then, the restriction of M to E' is called a* partial MSC *over A and P. The set of partial MSCs is denoted by* $\mathrm{p}\mathbb{MSC}(A, P)$.

In Figure 1, M'_0 is a partial MSC that is not an MSC. Notations such as $pid(e)$ carry over from MSCs to partial MSCs as expected. Let $M = (E, \lhd, \lambda, \mu) \in \mathrm{p}\mathbb{MSC}(A, P)$ be a partial MSC. By $MsgPar(M)$, we denote the set of $p \in P$ that occur as parameters in messages, i.e., those p, for which there is $a(p_1, \ldots, p_n) \in \mu(\lhd_{\mathsf{msg}})$ with $p \in \{p_1, \ldots, p_n\}$. For every $p \in P$ with $E_p \neq \emptyset$, there are a unique minimal and a unique maximal event in the total order $(E_p, \lhd^* \cap (E_p \times E_p))$, which we denote by $\min_p(M)$ and $\max_p(M)$, respectively.

We let $\mathit{Pids}(M) \stackrel{\text{def}}{=} \{p \in P \mid E_p \neq \emptyset\}$. By $\mathit{Free}(M) \stackrel{\text{def}}{=} \{p \in \mathit{Pids}(M) \mid E_{\mathsf{start}} \cap E_p = \emptyset\}$, we denote the set of *free* processes of M. Intuitively, free processes of a partial MSC M are processes that are not initiated in M. Moreover, $\mathit{Bnd}(M) \stackrel{\text{def}}{=} \mathit{Pids}(M) \setminus \mathit{Free}(M)$ denotes the set of *bound* processes. In Figure 1, we have $\mathit{Bnd}(M_0') = \{3\}$ and $\mathit{Free}(M_0') = \{0, 1, 2\}$.

Let $M = (E, \lhd, \lambda, \mu)$ and $M' = (E', \lhd', \lambda', \mu')$ be partial MSCs over A and P. The *concatenation* $M \circ M'$ glues identical processes together. It is defined if (i) $\mathit{Bnd}(M') \cap \mathit{Pids}(M) = \emptyset$, (ii) $\mathit{Free}(M') \neq \emptyset$, and (iii) $\mathit{Free}(M) = \emptyset$ implies $\mathit{Free}(M') \subseteq \mathit{Pids}(M)$. In that case, $M \circ M' \stackrel{\text{def}}{=} (\hat{E}, \hat{\lhd}, \hat{\lambda}, \hat{\mu})$ where $\hat{E} = E \uplus E'$, $\hat{\lhd}_{\mathsf{proc}} = \lhd_{\mathsf{proc}} \cup \lhd'_{\mathsf{proc}} \cup \{(\max_p(M), \min_p(M')) \mid p \in \mathit{Pids}(M) \cap \mathit{Pids}(M')\}$, $\hat{\lhd}_{\mathsf{msg}} = \lhd_{\mathsf{msg}} \cup \lhd'_{\mathsf{msg}}$, $\hat{\lhd}_{\mathsf{spawn}} = \lhd_{\mathsf{spawn}} \cup \lhd'_{\mathsf{spawn}}$, $\hat{\lambda} = \lambda \cup \lambda'$, and $\hat{\mu} = \mu \cup \mu'$.

Next we define a formalism to describe sets of MSCs. This is analogous to branching automata, but the transitions are labelled with partial MSCs.

Definition 3 (bHMSC). *A* branching high-level MSC (bHMSC) *over the set of message labels A is a tuple $\mathcal{H} = (L, X, L_{\text{init}}, L_{\text{acc}}, x_0, T)$ where L is the finite set of* locations, *$L_{\text{init}} \subseteq L$ is the set of* initial locations, *$L_{\text{acc}} \subseteq L$ is the set of* accepting locations, *X is the finite set of* registers *with* initial register *$x_0 \in X$, and T is the finite set of* transitions. *There are two types of transitions:*

- *A sequential transition is a triple $(\ell, M, \ell') \in L \times \mathrm{p}\mathbb{MSC}(A, X) \times L$, usually written $\ell \xrightarrow{M} \ell'$, such that $\mathit{Free}(M) \neq \emptyset$ and $\mathit{MsgPar}(M) \cap \mathit{Bnd}(M) = \emptyset$ (the latter guarantees an unambigous interpretation of message parameters).*
- *A fork-and-join transition is of the form $\ell \to \{(\ell_1, X_1, \ell_1'), \ldots, (\ell_n, X_n, \ell_n')\} \to \ell'$, where $n \geq 1$ is the* degree *of the transition, $\ell, \ell_1, \ldots, \ell_n, \ell_1', \ldots, \ell_n', \ell'$ are locations from L, and $X_1, \ldots, X_n$ are nonempty and pairwise disjoint subsets of X. It may also be depicted as*

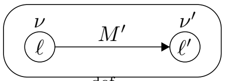
$$\begin{tikzcd}
& \ell_1 \arrow[r, dashed] & \ell_1' \arrow[dr, "X_1"] & \\
\ell \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots & \vdots & \ell' \\
& \ell_n \arrow[r, dashed] & \ell_n' \arrow[ur, "X_n"'] &
\end{tikzcd}$$

Fork-and-join transitions are similar to the split operator in [14]. At location ℓ, n subcomputations are started in $\ell_1, \ldots, \ell_n$, respectively, keeping only the register contents (pids) from $X_1, \ldots, X_n$. The other register contents are inaccessible until each subcomputaion i terminates at ℓ_i' (the registers as such may be used, but not their contents at ℓ). Then, the main computation resumes in ℓ', and registers in X_i adopt the final assignment from the i-th subcomputation.

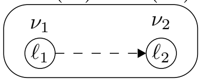
We associate MSCs with a bHMSC through the notion of runs, which we will define next after some preparation. A partial mapping $\nu : X \rightharpoonup \mathbb{P}$ is a *register assignment* if it is injective. The set of register assignments is denoted by $\mathcal{R}(X)$. For $\nu \in \mathcal{R}(X)$ and $Y \subseteq X$, we let $\nu_{\restriction Y} \stackrel{\text{def}}{=} \{x \mapsto \nu(x) \mid x \in \mathrm{dom}(\nu) \cap Y\}$. Given $\nu, \nu' \in \mathcal{R}(X)$ and an $M \in \mathrm{p}\mathbb{MSC}(A, X)$ that occurs in $\mathcal{H}$, we write $\nu \xrightarrow{M} \nu'$ (to be read as: M can be instantiated and performed at ν and yields ν') if

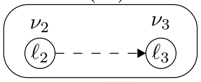
- $\mathit{Free}(M) \cup \mathit{MsgPar}(M) \subseteq \mathrm{dom}(\nu)$ (i.e., free processes can be instantiated),
- $\mathrm{dom}(\nu') = \mathrm{dom}(\nu) \cup \mathit{Bnd}(M)$, and ν and ν' coincide on $X \setminus \mathit{Bnd}(M)$ (i.e., registers remain unchanged unless they are overwritten for a new process),
- $\nu'(\mathit{Bnd}(M)) \cap \nu(X) = \emptyset$ (i.e., bound processes obtain fresh pids).

A run $G = (V, R, loc, reg, \rho)$ of the bHMSC $\mathcal{H}$ consists of a finite directed acyclic graph (V, R), $R \subseteq V \times V$, with a unique source node $in(G)$, a unique sink node $out(G)$, and labeling functions $loc : V \to L$, $reg : V \to \mathcal{R}(X)$, and $\rho : R \to 2^X \cup \mathrm{p}\mathbb{MSC}(A, \mathbb{P})$. The set of runs of $\mathcal{H}$ is defined inductively as follows:

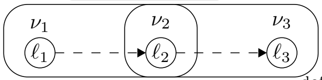
– Let $\nu, \nu' \in \mathcal{R}(X)$ be register assignments and let $\ell \xrightarrow{M} \ell'$ be a sequential transition such that $\nu \xrightarrow{M} \nu'$. Set $M' = \nu'(M)$, which we obtain from M by uniformly replacing x with $\nu'(x)$. Then, the graph $G =$ 

is a run of $\mathcal{H}$. We set $Pids(G) \stackrel{\text{def}}{=} \nu(X) \cup Pids(M')$ and $Bnd(G) \stackrel{\text{def}}{=} Bnd(M')$.

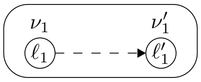
– Consider runs $G_1 =$ 

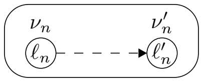
and $G_2 =$ 

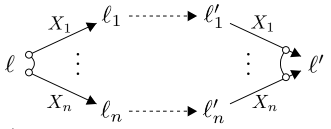
of $\mathcal{H}$.

If $Pids(G_1) \cap Bnd(G_2) = \emptyset$, then the graph $G =$ 

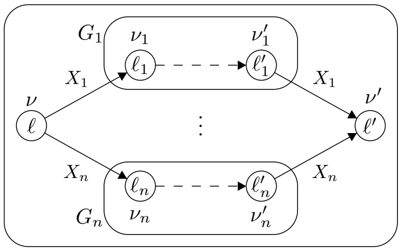
is a run of $\mathcal{H}$. We set $Pids(G) \stackrel{\text{def}}{=} Pids(G_1) \cup Pids(G_2)$ and $Bnd(G) \stackrel{\text{def}}{=} Bnd(G_1) \cup Bnd(G_2)$.

– For $n \geq 1$, let $G_1 =$ 

$, \ldots, G_n =$ 

be runs, 

be a fork-and-join transition, and $\nu, \nu' \in \mathcal{R}(X)$ be register assignments. Then, the graph

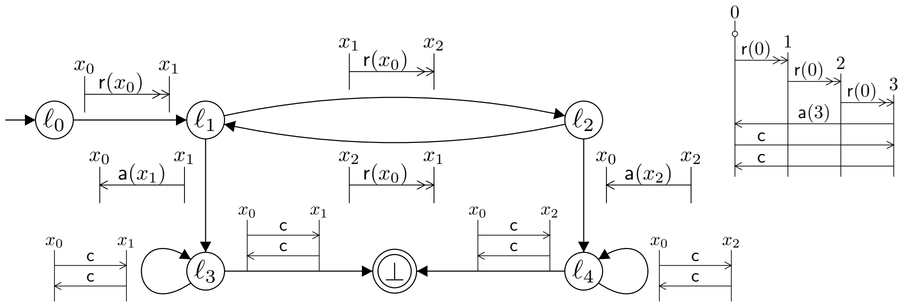
$G =$ 

is a run of $\mathcal{H}$ if $Bnd(G_i) \cap (\nu(X) \cup \bigcup_{j \neq i} Pids(G_j)) = \emptyset$ and $\nu_i = \nu_{\restriction X_i}$ for all $i \in \{1, \ldots, n\}$, and $\nu' = \nu_{\restriction X_0} \cup \bigcup_{i \in \{1,\ldots,n\}} (\nu_i')_{\restriction X_i}$ where $X_0 = X \setminus (X_1 \cup \ldots \cup X_n)$. We set $Pids(G) \stackrel{\text{def}}{=} \nu(X) \cup \bigcup_{i \in \{1,\ldots,n\}} Pids(G_i)$ and $Bnd(G) \stackrel{\text{def}}{=} \bigcup_{i \in \{1,\ldots,n\}} Bnd(G_i)$.

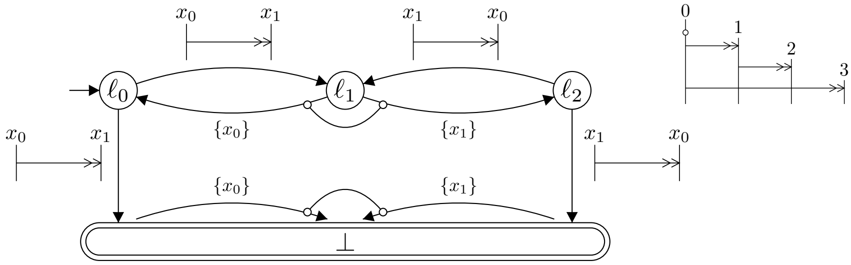
By choosing any enumeration $M_1, \ldots, M_n \in \mathrm{p}\mathbb{MSC}(A, \mathbb{P})$ of the partial MSCs occurring in G that respects the partial order induced by the edge relation R, we define $M(G) \stackrel{\text{def}}{=} M_1 \circ \ldots \circ M_n \in \mathrm{p}\mathbb{MSC}(A, \mathbb{P})$. Since, in a fork-and-join, subcomputations employ disjoint sets of pids, $M(G)$ is well defined and does not depend on the chosen enumeration. We call run G *accepting* if $loc(in(G)) \in L_{\text{init}}$,

$loc(out(G)) \in L_{\mathrm{acc}}$, and $reg(in(G)) = \{x_0 \mapsto p\}$ for some $p \in \mathbb{P}$. The language of $\mathcal{H}$ is $L(\mathcal{H}) \overset{\mathrm{def}}{=} \{ \overset{p}{\phi} \circ M(G) \mid G$ is an accepting run of $\mathcal{H}$ with $reg(in(G)) = \{x_0 \mapsto p\}\} \subseteq \mathbb{MSC}(A, \mathbb{P})$. Note that $L(\mathcal{H})$ is always closed.

Example 4. The bHMSC below models a peer-to-peer protocol. It has only sequential transitions and is defined over $A = \{\mathsf{r}, \mathsf{a}, \mathsf{c}\}$ (request, acknowledgment, communication) with $arity(\mathsf{r}) = arity(\mathsf{a}) = 1$ and $arity(\mathsf{c}) = 0$. The initial register is x_0. A request is forwarded to new processes along with the pid p of the initial process. At some point, a process acknowledges the request, sending its own pid q to the initial process. Processes p and q may then communicate and exchange messages. A generated MSC is depicted beside the bHMSC.



Example 5. The following bHMSC has one fork-and-join transition whose target state $\perp$ is the only final state. Due to the fork, registers can be used simultaneously at different places so that the generated MSCs have a tree-like structure.



Examples 4 and 5 represent important subclasses of bHMSCs, *sequential* and *join-free* bHMSCs, respectively, which we define in the following.

A bHMSC is called *sequential* if it contains only sequential transitions. Thus, the bHMSC from Example 4 is sequential.

Let $\mathcal{H} = (L, X, L_{\mathrm{init}}, L_{\mathrm{acc}}, x_0, T)$ be a bHMSC. By L_{seq}, L_{fork}, and $L_\perp$ we denote the sets of locations with outgoing sequential transitions, with outgoing fork-and-join transitions, and without outgoing transitions, respectively.

We say that bHMSC $\mathcal{H}$ is *join-free* if there is a distinguished location $\perp \in L$ such that $L_{\mathrm{acc}} = L_\perp = \{\perp\}$ and all fork-and-join transitions are of the form $\ell \to \{(\ell_1, X_1, \perp), \ldots, (\ell_n, X_n, \perp)\} \to \perp$. Thus, the bHMSCs from Examples 4 and 5 are *join-free*. The run of a join-free bHMSC may be viewed as a tree, as it can always be completed towards a run with a single target node. We will, therefore, consider that a fork-and-join transition is of the form

$$\begin{tikzcd} & \ell_1 \\ \ell \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \\ & \ell_n \end{tikzcd}$$

and rather call it a *fork transition.* Note that any bHMSC generating the MSCs $M(n)$ from Figure 1 is inherently not join-free. Moreover:

Lemma 6. *Join-free bHMSCs are more expressive than sequential bHMSCs.*

The first natural question to ask for a bHMSC $\mathcal{H}$ is whether $L(\mathcal{H}) \neq \emptyset$, i.e., the nonemptiness problem.

Theorem 7. *Nonemptiness of bHMSCs is EXPTIME-complete. It is already EXPTIME-hard for join-free bHMSCs. Nonemptiness of sequential bHMSCs is NP-complete.*

The proofs of the upper bounds use a notion of symbolic runs. EXPTIME-hardness is shown by a reduction from the intersection-nonemptiness problem for tree automata; for NP-hardness, we use a reduction from 3-CNF-SAT.

4 Dynamic Communicating Automata

In this section, we introduce an extension of the model of dynamic communicating automata as presented in [4]. A configuration of a DCA consists of several processes that can exchange messages through FIFO channels. A process can spawn new processes so that there is a priori no bound on the number of processes that participate in a system execution. In contrast to [4], we allow a message to contain process identities and receptions to be non-selective (i.e., a receiver may receive a message without knowing the sender).

Definition 8 (DCA). *A* dynamic communicating automaton (DCA) *over the ranked message alphabet A is a tuple $\mathcal{D} = (S, X, S_{\text{init}}, S_{\text{acc}}, \Delta)$ where S is a finite set of* states *with initial states $S_{\text{init}} \subseteq S$ and accepting states $S_{\text{acc}} \subseteq S$, X is a finite set of* registers*, and Δ is the set of* transitions*. A transition is of the form (s, α, s') where $s, s' \in S$, and α is an action, possibly a* send action *$!_x(a(x_1, \ldots, x_n))$, a* receive action *$?_y(a(y_1, \ldots, y_n))$, or a* spawn action *$x := \mathsf{spawn}(s, z)$, where $x, z \in X$, $y \in X \cup \{*\}$, $s \in S$, $a(x_1, \ldots, x_n) \in A(X \uplus \{\mathsf{self}\})$, and $a(y_1, \ldots, y_n) \in A(X \uplus \{-\})$ such that, for all $i, j \in \{1, \ldots, n\}$, $y_i = y_j \in X$ implies $i = j$.*

When a process executes $!_x(a(\overline{x}))$ with $\overline{x} = (x_1, \ldots, x_n)$, it sends a message to the process whose pid is stored in register x. The message consists of label a as well as $n = \mathit{arity}(a)$ many pids stored in registers $\overline{x}$ (or the sender's pid if $x_i = \mathsf{self}$). Executing $?_y(a(\overline{y}))$, a process receives a message from the process whose pid is stored in y (selective receive) or, in case $y = *$, from any process (non-selective receive). The message must be of the form $a(p_1, \ldots, p_n)$. In the resulting configuration, the receiving process updates its local registers $y_1, \ldots, y_n$ to $p_1, \ldots, p_n$, respectively, unless $y_i = -$. Finally, a process executing $x := \mathsf{spawn}(s, z)$ spawns a new process, whose fresh pid is henceforth stored in register x. The new process starts in state s. Its registers are a copy of the registers of the spawning process, except for z, which is set to the pid of the spawning process.

A *run* of DCA $\mathcal{D}$ on an MSC $M = (E, \lhd, \lambda, \mu) \in \mathbb{MSC}(A, \mathbb{P})$ is a pair (σ, τ), where $\sigma : E \to S$ and $\tau : E \to [X \rightharpoonup \mathbb{P}]$, respecting the following conditions:

- $\sigma_{init(M)} \in S_{\mathrm{init}}$,
- $\tau_{init(M)}$ is undefined everywhere,
- for all $e_1, e_2, f \in E$ with $e_1 \lhd_{\mathsf{proc}} e_2 \lhd_{\mathsf{spawn}} f$, the relation Δ contains a local transition $\sigma_{e_1} \xrightarrow{x := \mathsf{spawn}(s,y)} \sigma_{e_2}$ such that $\sigma_f = s$, $\tau_{e_2} = \tau_{e_1}[x \mapsto pid(f)]$, and $\tau_f = \tau_{e_1}[y \mapsto pid(e_1)]$, and
- for all $e_1, e_2, f_1, f_2 \in E$ with $e_1 \lhd_{\mathsf{proc}} e_2 \lhd_{\mathsf{msg}} f_2$ and $f_1 \lhd_{\mathsf{proc}} f_2$, the relation Δ contains transitions $\sigma_{e_1} \xrightarrow{!_x(a(x_1,\ldots,x_n))} \sigma_{e_2}$ and $\sigma_{f_1} \xrightarrow{?_y(a(y_1,\ldots,y_n))} \sigma_{f_2}$ such that $\{x, x_1, \ldots, x_n\} \subseteq \mathrm{dom}(\tau_{e_1}) \cup \{\mathsf{self}\}$, $\tau_{e_2} = \tau_{e_1}$, $\tau_{e_1}(x) = pid(f_1)$, $\big(y = *$ or $\tau_{f_1}(y) = pid(e_1)\big)$, and, letting $p_i = \begin{cases} \tau_{e_1}(x_i) & \text{if } x_i \in X \\ pid(e_1) & \text{if } x_i = \mathsf{self}\,, \end{cases}$ we have
$$\mu(e_2, f_2) = a(p_1, \ldots, p_n) \text{ and } \tau_{f_2}(z) = \begin{cases} p_i & \text{if } z = y_i \\ \tau_{f_1}(z) & \text{if } z \notin \{y_1, \ldots, y_n\}\,. \end{cases}$$

Here, σ_e and τ_e denote $\sigma(e)$ and $\tau(e)$, respectively. Moreover, $\tau_e[x \mapsto p]$ is the partial mapping that maps x to p and coincides with τ_e on all other arguments.

The run (σ, τ) is accepting if $\sigma_e \in S_{\mathrm{acc}}$ for all $e \in \{\max_p(M) \mid p \in \mathit{Pids}(M)\}$. By $L(\mathcal{D})$, we denote the set of MSCs M over A and $\mathbb{P}$ such that there is an accepting run of $\mathcal{D}$ on M. Note that $L(\mathcal{D})$ is closed, i.e., $L(\mathcal{D}) = [L(\mathcal{D})]$. Nonemptiness is undecidable for CA, and consequently also for DCA.

There are languages L that are not the language of a DCA, but for which there is a DCA *implementing* them up to some refinement. The refinement allows a DCA to attach more information to a message than the specification provides, for example additional pids. This is formalized as follows. Let A, B be ranked alphabets and let $h : B \to A$. We say that the pair (B, h) is a refinement of A if, for all $b \in B$, $\mathit{arity}(h(b)) \leq \mathit{arity}(b)$. We can extend h to a mapping $h : \mathbb{MSC}(B, \mathbb{P}) \to \mathbb{MSC}(A, \mathbb{P})$ as follows: for an MSC $M = (E, \lhd, \lambda, \mu) \in \mathbb{MSC}(B, \mathbb{P})$, we let $h(M) = (E, \lhd, \lambda, \mu') \in \mathbb{MSC}(A, \mathbb{P})$ where $\mu'(e, f) = h(b)(p_1, \ldots, p_{\mathit{arity}(h(b))})$ whenever $\mu(e, f) = b(p_1, \ldots, p_n)$. The mapping is then further extended to sets of MSCs as expected.

Definition 9 (realizable, implementable). *We call a set $L \subseteq \mathbb{MSC}(A, \mathbb{P})$* realizable *if $[L] = L(\mathcal{D})$ for some DCA $\mathcal{D}$. We say that L is* implementable *if there are a refinement (B, h) of A and a DCA $\mathcal{D}$ over B such that $[L] = h(L(\mathcal{D}))$.*

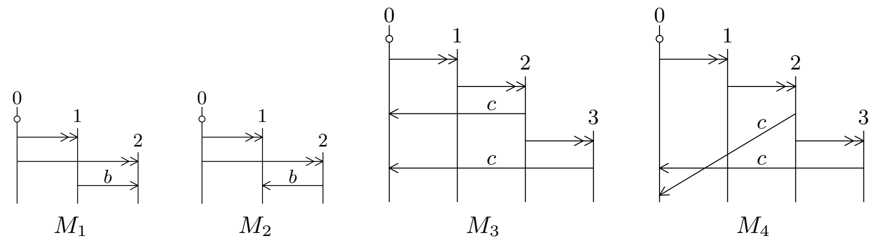


Fig. 2. Realizability vs. Implementability

For both realizability and implementability, it is necessary that the sender p of a message knows the receiver q at the time of sending, i.e., q should be stored in some register of p. Note that this aspect does not arise in simple CA.

Example 10. The MSC language $\{M_1\}$ (see Figure 2) is not implementable, as process 1 does not know 2 when sending message b. However, $\{M_2\}$ is implementable (and even realizable), as 2 may know 1: when spawning 2, process 0 can communicate the pid 1 to 2. The language $\{M_3\}$ is not realizable: as process 0 does neither know 2 nor 3 when it receives the messages, it has to use a non-selective receive. But then, the DCA also accepts M_4. On the other hand, $\{M_3, M_4\}$ is realizable. However, $\{M_3\}$ and $\{M_4\}$ are implementable by refining the messages from 2 and 3.

5 Executability

An accepting run of a bHMSC generates an MSC. However, this MSC need not be implementable always, as Example 10 shows. Unfortunately, implementability (and also realizability) is undecidable for bHMSCs, which follows from undecidability for HMSCs over a fixed finite set of processes [12,1].

Theorem 11 (cf. [12,1]). *Implementability and realizability of bHMSCs are undecidable. This already holds for sequential bHMSCs.*

We now focus on implementability and introduce an effective necessary criterion, called executability: every sender in a generated MSC should be "aware of" the receiver and the processes whose pids are used as message parameters.

Given an MSC M, a process q and an event e of M, we write $q \rightsquigarrow_M e$ if there is a path from the minimal event $\min_q(M)$ of q to e in M. This path might involve the reversal of the spawn edge that started q. That is, $q \rightsquigarrow_M e$ if $(\min_q(M), e) \in (\lhd \cup \lhd_{\mathsf{spawn}}^{-1})^*$. Intuitively, $q \rightsquigarrow_M e$ indicates that the process executing e is aware of process q. Next, we formally define executability of MSCs.

Definition 12 (executability). *Let $M \in \mathbb{MSC}(A, \mathbb{P})$. A message $(e, f) \in \lhd_{\mathsf{msg}}$ of M with message contents $a(p_1, \ldots, p_n)$ is* executable *if $q \rightsquigarrow_M e$, for every*

$q \in \{pid(f), p_1, \ldots, p_n\}$. Moreover, M is executable *if each of its messages is executable. Finally, a bHMSC $\mathcal{H}$ is* executable *if each MSC from $L(\mathcal{H})$ is executable.*

For example, in Figure 2, M_2, M_3, M_4 are executable, while M_1 is not. Let M be an MSC and $\mathcal{H}$ be a bHMSC. One can verify that 1) M is executable iff $\{M\}$ is implementable, and 2) $\mathcal{H}$ is executable if it is implementable (while the converse might fail). Unlike implementability, executability is decidable:

Theorem 13. *Executability of bHMSCs is EXPTIME-complete. Moreover, the lower bound already holds for bHMSCs that are join-free.*

The lower bound is deduced from the lower bound of the nonemptiness problem (Theorem 7). For the upper bound, we abstract the knowledge of processes by a finite number of *awareness relations*, so as to work over symbolic runs.

6 Implementing Guarded Join-Free bHMSCs

We identify a subclass of bHMSCs for which executability and implementabiliy coincide. *Guarded* bHMSCs are based on the notion of a leader process, which determines the next transition to be taken in a bHMSC. They are an adaptation of locality from [10]. For $M = (E, \lhd, \lambda, \mu) \in \mathrm{p}\mathbb{MSC}(A, X)$, $Y \subseteq X$, and $x \in X$, we write $Y \preceq_M x$ if $x \in \mathit{Pids}(M) \cap Y$ and, for all $y \in \mathit{Pids}(M) \cap Y$, $\max_y(M) \lhd^* \max_x(M)$. Intuitively, all processes in $\mathit{Pids}(M) \cap Y$ terminate before x.

Definition 14 (guarded). *A join-free bHMSC $\mathcal{H} = (L, X, L_{\mathrm{init}}, L_{\mathrm{acc}}, x_0, T)$ is called* guarded *if $L = L_{\mathrm{seq}} \uplus L_{\mathrm{fork}} \uplus \{\bot\}$, $L_{\mathrm{init}} \subseteq L_{\mathrm{seq}}$, and there is a mapping leader $: L_{\mathrm{seq}} \to X$ such that*

1. *for all partial MSCs $M = (E, \lhd, \lambda, \mu) \in \mathrm{p}\mathbb{MSC}(A, X)$ that occur in $\mathcal{H}$, $(E, \lhd^*)$ has a unique minimal element e; we let first$(M) \overset{def}{=} pid(e)$,*
2. *for all sequential transitions $\ell \xrightarrow{M} \ell'$, it holds leader$(\ell)$ = first(M), and, if $\ell' \in L_{\mathrm{seq}}$, also $X \preceq_M$ leader(ℓ'), and*
3. *for all transition patterns* $$\begin{tikzcd}[row sep=small] & & \ell_1 \\ \ell \arrow[r, "M"] & \ell' \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \\ & & \ell_n \end{tikzcd}$$ *and all $i \in \{1, \ldots, n\}$, we have $\ell_i \in L_{\mathrm{seq}}$ and $X_i \preceq_M$ leader(ℓ_i).*

Example 15. The bHMSCs from Examples 4 and 5 are both guarded.

Theorem 16. *A guarded join-free bHMSC is implementable if and only if it is executable. Moreover, if it is implementable, an equivalent DCA can be constructed in exponential time.*

Towards an implementation of a given guarded join-free bHMSC $\mathcal{H}$, we first enrich locations of $\mathcal{H}$ with awareness relations (in the same spirit as in the proof of Theorem 13). Then, we rely on techniques employed in the context of a bounded number of processes [11,10], to build a DCA (together with a refinement) that recognizes $L(\mathcal{H})$.

Note that guardedness does not yield better complexities:

Theorem 17. *Nonemptiness and executability of guarded join-free bHMSCs are both EXPTIME-complete.*

7 Future Work

In future work, we aim at finding classes of bHMSCs for which executability and implementability coincide and that are not necessarily join-free or guarded (e.g., by transferring concepts like fork-acyclicity from branching automata to bHMSCs). Moreover, connections with the π-calculus [19] should be explored.

Acknowledgments. We thank the anonymous reviewers as well as Martin Schuster and Thomas Zeume for their helpful comments.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theoretical Computer Science 331(1), 97–114 (2005)
2. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. Logical Methods in Computer Science 7(4) (2011)
3. Bollig, B., Cyriac, A., Hélouët, L., Kara, A., Schwentick, T.: Dynamic Communicating Automata and Branching High-Level MSCs. Research Report LSV-12-20, LSV (November 2012)
4. Bollig, B., Hélouët, L.: Realizability of Dynamic MSC Languages. In: Ablayev, F., Mayr, E.W. (eds.) CSR 2010. LNCS, vol. 6072, pp. 48–59. Springer, Heidelberg (2010)
5. Borgström, J., Gordon, A., Phillips, A.: A chart semantics for the Pi-calculus. Electronic Notes in Theoretical Computer Science 194(2), 3–29 (2008)
6. Bozzelli, L., La Torre, S., Peron, A.: Verification of well-formed communicating recursive state machines. Theoretical Computer Science 403(2-3), 382–405 (2008)
7. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM 30(2) (1983)
8. Buscemi, M.G., Sassone, V.: High-Level Petri Nets as Type Theories in the Join Calculus. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 104–120. Springer, Heidelberg (2001)
9. Genest, B., Kuske, D., Muscholl, A.: A Kleene theorem and model checking algorithms for existentially bounded communicating automata. Information and Computation 204(6), 920–956 (2006)

10. Genest, B., Muscholl, A., Seidl, H., Zeitoun, M.: Infinite-state high-level MSCs: Model-checking and realizability. Journal of Computer and System Sciences 72(4), 617–647 (2006)
11. Hélouët, L., Jard, C.: Conditions for synthesis of communicating automata from HMSCs. In: Proceedings of FMICS 2000, pp. 203–224. Springer (2000)
12. Henriksen, J.G., Mukund, M., Narayan Kumar, K., Sohoni, M.A., Thiagarajan, P.S.: A theory of regular MSC languages. Inf. Comput. 202(1), 1–38 (2005)
13. ITU-TS: ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva (February 2011)
14. Leucker, M., Madhusudan, P., Mukhopadhyay, S.: Dynamic Message Sequence Charts. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 253–264. Springer, Heidelberg (2002)
15. Lodaya, K., Weil, P.: Series-parallel languages and the bounded-width property. Theoretical Computer Science 237(1-2), 347–380 (2000)
16. Lodaya, K., Weil, P.: Rationality in algebras with a series operation. Information and Computation 171(2), 269–293 (2001)
17. Lohrey, M.: Realizability of high-level message sequence charts: closing the gaps. Theoretical Computer Science 309(1-3), 529–554 (2003)
18. Meyer, R.: On Boundedness in Depth in the π-Calculus. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Proceedings of IFIP TCS 2008, vol. 273, pp. 477–489. Springer, Boston (2008)
19. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Information and Computation 100(1), 1–40 (1992)
20. Morin, R.: Recognizable Sets of Message Sequence Charts. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 523–534. Springer, Heidelberg (2002)
21. Segoufin, L.: Automata and Logics for Words and Trees over an Infinite Alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)

Visibly Pushdown Automata: Universality and Inclusion via Antichains

Véronique Bruyère[1], Marc Ducobu[1,⋆], and Olivier Gauwin[2]

[1] University of Mons, UMONS, Belgium
[2] University of Bordeaux, LaBRI, France

Abstract. Visibly pushdown automata (VPAs), introduced by Alur and Madhusudan in 2004, are a useful formalism in various contexts, such as expressing and checking properties on control flows of programs, or on XML documents. In this context, we propose efficient antichain-based algorithms to check universality and inclusion of VPAs. Whereas the computation complexity is known to be ExpTime-complete for both problems, we show how antichains can avoid explicit determinization and save computations. The approach is extended to hedge automata. We implement the proposed algorithms in a prototype tool and conduct experiments on randomly generated VPAs. We show that, on numerous instances, our algorithms outperform other VPA tools.

1 Introduction

The model-checking framework provided many successful tools for decades, starting from the seminal work of Büchi. A lot of them rely on the links between logics used to express properties on words, and automata allowing to check them. Some of these results have been adapted to trees, and more recently to words with a nesting structure.

Visibly pushdown automata (VPAs) have been introduced to process such words with nesting [2]. VPAs are similar to pushdown automata, but operate on a partitioned alphabet: a given letter is associated with one action (push or pop), and thus cannot push when firing a transition, and pop when firing another. Such automata were introduced to express and check properties on control flows of programs, where procedure calls push on the stack, and returns pop [15]. They are also suitable to express properties on XML documents [14]. These documents are usually represented as trees, and serialized as a sequence of opening and closing tags, also called the *linearization* of this document, or its corresponding *XML stream.*

The model-checking framework with VPAs is confronted to the computational hardness of testing universality and inclusion. These two problems are ExpTime-complete on non-deterministic VPAs, due to the expensive determinization step [2]. Non-determinism naturally arises when automata are obtained from logic formulas, as for instance XPath expressions with descendant axis [11].

⋆ The second author is supported by a grant from FRIA.

In this paper we propose antichain-based algorithms for deciding universality and inclusion of VPAs. We use *antichains* to get smaller objects to manipulate and to avoid an explicit determinization step. Recently, antichains have been successfully applied to decision problems related to non-deterministic automata: universality and inclusion for finite word automata [7], and for non-deterministic bottom-up tree automata [4]. This latter work also provides a way to check universality and inclusion over regular unranked tree languages, by encoding unranked trees into binary trees. We plan to investigate this procedure in the near future. Some simulation relations are also known on unranked trees [18,1,8] but it is unclear whether they can help for our problems.

Nguyen [16] proposed an algorithm for testing the universality of VPAs. This algorithm simultaneously performs an on-the-fly determinization and reachability checking by $\mathcal{P}$-automaton. The notion of $\mathcal{P}$-automaton proposed in [10] provides a symbolic technique to compute the sets of all reachable configurations of a VPA. This algorithm has been later improved by Nguyen and Ohsaki [17] by introducing antichains over transitions of $\mathcal{P}$-automata, in a way to generate the smallest amount of reachable configurations. Our algorithms for universality are alternative to this one since we do not use the regularity property of the set of reachable configurations. When observed on trees, their approach follows forward steps on the linearization of trees, while our approach is bottom-up on the structure of trees.

In [12], the authors provide solutions for checking universality and inclusion of VPAs over finite and infinite words. They avoid the determinization and complementation steps, and use instead Ramsey-based universality- and inclusion-checking algorithms. Their algorithms do not seem to use antichains.

The paper is structured as follows. In Section 2 we define unranked trees and visibly pushdown automata seen as trees acceptors. In Section 3, we detail our antichain-based algorithm for checking universality of VPAs, together with several optimizations. Section 4 contains extensions of this algorithm to general VPAs and hedge automata. It also proposes an antichain-based algorithm for testing the inclusion of VPAs. Section 5 is devoted to the experiments and comparisons with other prototype tools. The long version of this paper is available on the third author's website.

2 Preliminaries

2.1 Unranked Trees

We here recall the standard definition of unranked trees, as provided for instance in [6]. Let Σ be a finite *alphabet*, and Σ^* (resp. Σ^+) be the set of all words (resp. non empty words) over Σ. The empty word is denoted by ϵ.

An *unranked tree* t over Σ is a partial function $t : (\mathbb{N} \setminus \{0\})^* \to \Sigma$ such that the domain is non-empty, finite and prefix-closed. The domain is denoted by $nodes(t)$ and contains the *nodes* of the tree t, with the root being the empty word ϵ. The function t labels each node p with a letter $t(p)$ of Σ. The set of all unranked trees over Σ is denoted by T_Σ. A *hedge* h over Σ is a finite sequence (empty or not) of unranked trees over Σ. The empty hedge is denoted by ϵ, and the set of all hedges over Σ is denoted by H_Σ.

Given a tree t, the *subtree* of t rooted at node p of t is the tree denoted by $t_{|p}$, which domain is the set of nodes p' such that $pp' \in \mathit{nodes}(t)$ and verifying $t_{|p}(p') = t(pp')$. For a given node $p \in \mathit{nodes}(t)$, we call *children* of p the nodes $pi \in \mathit{nodes}(t)$ for $i \in \mathbb{N} \setminus \{0\}$, and use the usual definitions for parents, ancestors and descendants. The *height* of a tree, and more generally of a hedge, is the length of its longest branch (with the length being the number of nodes).

Trees can be described by well-balanced words which correspond to a depth-first traversal of the tree. An opening tag is used to notice the arrival on a node and a closing tag to notice the departure from a node. For each $a \in \Sigma$, let a itself represent the opening tag and $\overline{a}$ the related closing tag. The *linearization* $[t]$ of $t \in T_\Sigma$ is the *well-balanced* word over $\Sigma \cup \overline{\Sigma}$, with $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$, inductively defined by: $[t] = a\ [t_{|1}] \cdots [t_{|n}]\ \overline{a}$, with $a = t(\epsilon)$ and the root has n children.

2.2 Visibly Pushdown Automata

Visibly pushdown automata (VPAs, [2,3]) are pushdown automata operating on a partitioned alphabet where only call symbols can push, return symbols can pop, and internal symbols can do transitions without considering the stack. For clarity we only consider languages of unranked trees, so we use VPAs as *unranked tree acceptors*, operating on their linearization [13].[1]

Definition 1. *A visibly pushdown automaton $\mathcal{A}$ over a finite alphabet Σ is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ where Q is a finite set of states containing initial states $Q_i \subseteq Q$ and final states $Q_f \subseteq Q$, a finite set Γ of stack symbols, and a finite set Δ of rules. Each rule in Δ is of the form $q \xrightarrow{a:\gamma} q'$ with $a \in \Sigma \cup \overline{\Sigma}$, $q, q' \in Q$, and $\gamma \in \Gamma$.*

A *configuration* of a VPA $\mathcal{A}$ is a pair (q, σ) where $q \in Q$ is a state and $\sigma \in \Gamma^*$ a stack content. A configuration is *initial* (resp. *final*) if $q \in Q_i$ (resp. $q \in Q_f$) and $\sigma = \epsilon$. For $a \in \Sigma \cup \overline{\Sigma}$, we write $(q, \sigma) \xrightarrow{a} (q', \sigma')$ if there is a transition $q \xrightarrow{a:\gamma} q'$ in Δ verifying $\sigma' = \gamma \cdot \sigma$ if $a \in \Sigma$, and $\sigma = \gamma \cdot \sigma'$ if $a \in \overline{\Sigma}$.

A *run* of a VPA $\mathcal{A}$ on a word $a_1 \cdots a_n \in (\Sigma \cup \overline{\Sigma})^*$ is a sequence of configurations (q_i, σ_i), $0 \leq i \leq n$, such that $(q_{i-1}, \sigma_{i-1}) \xrightarrow{a_i} (q_i, \sigma_i)$ for all i. It is denoted by $(q_0, \sigma_0) \xrightarrow{a_1 \cdots a_n} (q_n, \sigma_n)$. It is *accepting* if $a_1 \cdots a_n$ is the linearization of a tree $t \in T_\Sigma$, (q_0, σ_0) is initial, and (q_n, σ_n) is final. A tree $t \in T_\Sigma$ is *accepted* by $\mathcal{A}$ if there is an accepting run on its linearization $[t]$. The set of accepted trees is called the *language* of $\mathcal{A}$ and is written $L(\mathcal{A})$.

A VPA $\mathcal{A}$ is said *universal* if it accepts all trees, i.e. $L(\mathcal{A}) = T_\Sigma$.[2] Our objective in the next section, is to propose efficient algorithms for testing universality of

[1] These VPAs used as tree acceptors do not use internal symbols. The algorithms we design for them can easily be adapted to usual VPAs, as explained in Section 4.2.

[2] This notion of universality is different from the one proposed for usual VPAs, where a VPA is universal if it accepts all words of $(\Sigma \cup \overline{\Sigma})^*$.

VPAs. A standard method to check universality of a VPA is to determinize it, complement it, and check for emptiness. As determinization is in exponential time for VPAs, the universality problem is ExpTime-complete. Our algorithms aim at avoiding this exponential blow-up on numerous instances.

3 Checking Universality

In our approach for checking universality of VPAs, the main idea is to find as fast as possible a tree which linearization is rejected by the automaton (if it exists), without computing an explicit determinization of the VPA. Antichains will limit the computations. Proofs are given in the long version of the paper.

3.1 Accessibility Relation

Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ be a VPA, and t be a tree over Σ. We define the *accessibility relation* $Acc(t) \subseteq Q \times Q$ as the set

$$Acc(t) = \{(q, q') \in Q \times Q \mid (q, \epsilon) \xrightarrow{[t]} (q', \epsilon)\}.$$

Note that in this paper, accessibility means *accessibility through the linearization of a tree*, as opposed to the accessibility between configurations of the VPA. With this notation, a tree t is accepted by $\mathcal{A}$ iff $Acc(t) \cap Q_i{\times}Q_f \neq \emptyset$.

Let us show how the accessibility relation $Acc(t)$ can be computed for all $t \in T_\Sigma$. In this aim, we need to introduce the *Post* operator. For a set $\mathscr{R}$ of relations $r \subseteq Q \times Q$, let $\mathscr{R}^*$ denote the set of all relations obtained by composing elements of $\mathscr{R}$: $\mathscr{R}^* = \{r_1 \circ r_2 \circ \cdots \circ r_n \mid n \geq 0 \text{ and } r_i \in \mathscr{R} \text{ for all } 1 \leq i \leq n\}$. In particular $\mathscr{R}^*$ contains the identity relation id_Q over Q, obtained when $n = 0$.

Definition 2. *Given $r \in \mathscr{R}$ and $a \in \Sigma$, let*

$$Post_a(r) = \{(p, p') \in Q \times Q \mid \exists (q, q') \in r, p \xrightarrow{a:\gamma} q \in \Delta, q' \xrightarrow{\overline{a}:\gamma} p' \in \Delta\}.$$

For $\mathscr{R}$ a set of relations over Q, let

$$Post(\mathscr{R}) = \{Post_a(r) \mid a \in \Sigma, r \in \mathscr{R}^*\} \cup \mathscr{R}$$

and $Post^*(\mathscr{R}) = \cup_{i \geq 0} Post^i(\mathscr{R})$ such that $Post^0(\mathscr{R}) = \mathscr{R}$, and for all $i > 0$, $Post^i(\mathscr{R}) = Post(Post^{i-1}(\mathscr{R}))$.

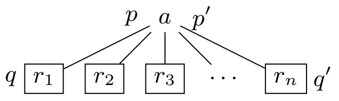
We illustrate the definition of *Post* in Figure 1. The following lemmas relate these operators with the accessibility relation.

Lemma 3. *Let $t \in T_\Sigma$ be such that its root is a node with n children that is labeled by a. Let $r_i = Acc(t_{|i})$ for $1 \leq i \leq n$. Then $Acc(t) = Post_a(r_1 \circ \cdots \circ r_n)$.*

Lemma 4. *$Post^i(\emptyset) = \{Acc(t) \mid t$ is a tree with height $\leq i\}$.*

The next proposition is an immediate consequence of Lemmas 3 and 4.

Proposition 5. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ be a VPA. Then $\mathcal{A}$ is universal iff $\forall r \in Post^*(\emptyset)$, $r \cap Q_i{\times}Q_f \neq \emptyset$.*



$$p \xrightarrow{a:\gamma} q \in \Delta,$$
$$q' \xrightarrow{\overline{a}:\gamma} p' \in \Delta,$$
$$(q,q') \in r_1 \circ r_2 \circ \cdots \circ r_n$$

Fig. 1. $(p,p') \in Post_a(r_1 \circ \cdots \circ r_n)$

```
Function Universality($\mathcal{A}$)
  $\mathscr{R} \leftarrow \emptyset$;
  $\mathscr{R}^* \leftarrow \{id_Q\}$;
  repeat
      $\mathscr{R}_{new} \leftarrow \{Post_a(r) \mid a \in \Sigma, r \in \mathscr{R}^*\} \setminus \mathscr{R}$;
      if $\exists r \in \mathscr{R}_{new} : r \cap Q_i \times Q_f = \emptyset$ then
          return False ;                                   /* Not universal */
      end
      $\mathscr{R} \leftarrow \mathscr{R} \cup \mathscr{R}_{new}$ ;
      $\mathscr{R}' \leftarrow \mathscr{R}_{new} \setminus \mathscr{R}^*$ ;
      if $\mathscr{R}' \neq \emptyset$ then
          $\mathscr{R}^* \leftarrow$ CompositionClosure($\mathscr{R}^*$,$\mathscr{R}'$);
      end
  until $\mathscr{R}' = \emptyset$ ;
  return True ;                                            /* Universal */
```

3.2 Algorithm

We are now able to propose an algorithm to check the universality of VPAs. With Function 1, the set $Post^*(\emptyset)$ is computed incrementally and the universality test is performed thanks to Proposition 5. At step i, the variable $\mathscr{R}$ is used for $Post^i(\emptyset)$, and the variable $\mathscr{R}^*$ contains its closure by composition. We compute $\mathscr{R}^*$ with Function CompositionClosure, and then potential new relations with $\{Post_a(r) \mid a \in \Sigma, r \in \mathscr{R}^*\}$. The algorithm always stops, either because a relation proving non-universality is found, or no new relation can be produced.

Let us detail Function CompositionClosure($\mathscr{R}^*$,$\mathscr{R}'$) which computes the set $(\mathscr{R}^* \cup \mathscr{R}')^*$. In Function 2, we show how to compute this set without recomputing $\mathscr{R}^*$ from $\mathscr{R}$. Initially, *Relations* is equal to $\mathscr{R}^*$ and will be equal to $(\mathscr{R}^* \cup \mathscr{R}')^*$ at the end of the computation. *ToProcess* contains the relations that can produce new relations by composition with an element of *Relations*.

Proposition 6. *Given $\mathscr{R}^*$ and $\mathscr{R}'$, Function 2 computes $(\mathscr{R}^* \cup \mathscr{R}')^*$.*

3.3 Antichain-Based Optimization

In this section we explain how to use the concept of antichain for saving computations. We show that it is sufficient to only compute the $\subseteq$-minimal elements of $Post^*(\emptyset)$ for checking universality.

Function `CompositionClosure`($\mathscr{R}^*, \mathscr{R}'$)

```
Relations ← $\mathscr{R}^*$;
ToProcess ← $\mathscr{R}'$;
while ToProcess $\neq \emptyset$ do
    rel ← Pop(ToProcess);
    $\mathscr{R}^* \leftarrow \mathscr{R}^* \cup \{rel\}$;
    NewRelations ← $\emptyset$;
    for $r \in$ Relations do
        NewRelations ← NewRelations $\cup \{r \circ rel, rel \circ r\}$;
    end
    ToProcess ← ToProcess $\cup$ (NewRelations $\setminus$ Relations);
    Relations ← Relations $\cup$ NewRelations;
end
return Relations
```

Consider the set $2^{Q \times Q}$ of all binary relations over Q. This set is equipped with the $\subseteq$ operator such that $r \subseteq r'$ iff $(q, q') \in r \Rightarrow (q, q') \in r'$. An *antichain* $\mathscr{R}$ of relations over Q is a set of pairwise incomparable relations with respect to $\subseteq$. Given a set $\mathscr{R}$ of relations, we denote by $\lfloor \mathscr{R} \rfloor$ the $\subseteq$-*minimal* elements of $\mathscr{R}$.

Definition 7. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ be a VPA, and $\mathscr{R}$ be a set of relations over Q. Let $Post^*_{min}(\mathscr{R}) = \cup_{i \geq 0} Post^i_{min}(\mathscr{R})$ such that $Post^0_{min}(\mathscr{R}) = \lfloor \mathscr{R} \rfloor$, and for all $i > 0$, $Post^i_{min}(\mathscr{R}) = \left\lfloor Post(Post^{i-1}_{min}(\mathscr{R})) \right\rfloor$.*

Lemma 8. *Given $\mathscr{R}$ a set of relations over Q, for all $r \in Post^*(\mathscr{R})$, there exists $r' \in Post^*_{min}(\mathscr{R})$ such that $r' \subseteq r$.*

We have the next counterpart of Proposition 5.

Proposition 9. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_i, Q_f, \Delta)$ be a VPA. Then $\mathcal{A}$ is universal iff $\forall r \in Post^*_{min}(\emptyset), r \cap Q_i \times Q_f \neq \emptyset$.*

Function 3 checks whether a VPA is universal by computing incrementally $Post^*_{min}(\emptyset)$. It is an adaptation of Function 1. Notice that we can compute $\lfloor Post(\mathscr{R}) \rfloor$ as $\lfloor \{Post_a(r) \mid a \in \Sigma, r \in \lfloor \mathscr{R}^* \rfloor\} \cup \mathscr{R} \rfloor$ (we limit the computation to $r \in \lfloor \mathscr{R}^* \rfloor$). The set $\lfloor \mathscr{R}^* \rfloor$ is denoted by $\mathscr{R}^*_{min}$ in the algorithm.

3.4 Other Optimizations

Functions 2 and 3 can be optimized in several directions. The optimized algorithm is given in the long version of the paper.

Witness of Non Universality - As soon as a new relation r is yielded (via the *Post* operator or the composition of two relations), the emptiness of its intersection with $Q_i \times Q_f$ is *immediately* tested, to check whether it is a witness of non universality.

In Function 2, the relation *rel* is composed by all relations $r \in \mathscr{R}$. We consider several implementations of *ToProcess*, with the aim that Pop (*ToProcess*) returns

```
Function UniversalityATC($\mathcal{A}$)
  $\mathscr{R} \leftarrow \emptyset$;
  $\mathscr{R}^*_{min} \leftarrow \{id_Q\}$;
  repeat
      $\mathscr{R}_{new} \leftarrow \lfloor\{Post_a(r) \mid a \in \Sigma, r \in \mathscr{R}^*_{min}\}\rfloor \setminus \mathscr{R}$;
      if $\exists r \in \mathscr{R}_{new} : r \cap Q_i{\times}Q_f = \emptyset$ then
          return False ;                                        /* Not universal */
      end
      $\mathscr{R} \leftarrow \lfloor\mathscr{R} \cup \mathscr{R}_{new}\rfloor$;
      $\mathscr{R}' \leftarrow \mathscr{R}_{new} \setminus \mathscr{R}^*_{min}$;
      if $\mathscr{R}' \neq \emptyset$ then
          $\mathscr{R}^*_{min} \leftarrow \lfloor$CompositionClosure($\mathscr{R}^*_{min}, \mathscr{R}'$)$\rfloor$;
      end
  until $\mathscr{R}' = \emptyset$ ;
  return True ;                                                 /* Universal */
```

a relation *rel* that is the *most promising* to yield a witness of non universality of small size. We try to take the sizes of the domain and codomain of the relations into account. Indeed, the compositions $r \circ rel$, $rel \circ r$ should be more incline to not intersect $Q_i{\times}Q_f$ when domains and codomains of *rel* are small. We also try to implement *ToProcess* as a queue. We note that no method appears to be better. We keep the best results in the experiments.

Data Structures - Efficient data structures are used both for the relations and the antichains. A relation is stored as an array of *bit-vectors.* In this way the composition is computed efficiently using bit-operations, as well as its domain and codomain.

A *hash table* is used to store an antichain, such that relations with different domain or codomain are stored in different lists. In this way, comparing a new relation r with the elements of the antichain is made more efficient, by limiting the comparison of r with elements of the same domain and codomain.

To compute $\lfloor\mathscr{R}^*\rfloor$ in Function 3, we first make a call to Function Composition-Closure, and then keep the $\subseteq$-minimal elements of the result. One optimization is, at *each* step of the CompositionClosure computation, to only consider the minimal elements.

4 Extensions

In this section, we propose several extensions. We first show how to adapt our antichain-based algorithm for testing the inclusion of two VPAs. Then we extend our algorithm for testing universality of the original VPA model [2], accepting words with internal actions and pending calls or returns. Finally, a third extension is proposed for checking universality of hedge automata.

4.1 Checking Inclusion

An antichain-based algorithm for testing the inclusion of two VPAs can be designed, following the same ideas as for testing universality. We here give the main ideas for VPAs accepting linearizations of trees. For two VPAs $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \Gamma_{\mathcal{A}}, Q_{i,\mathcal{A}}, Q_{f,\mathcal{A}}, \Delta_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \Gamma_{\mathcal{B}}, Q_{i,\mathcal{B}}, Q_{f,\mathcal{B}}, \Delta_{\mathcal{B}})$ over the same alphabet Σ, we have $L(\mathcal{A}) \nsubseteq L(\mathcal{B})$ iff there exists in $\mathcal{A}$ an accepting run on some word w such that all runs on w in $\mathcal{B}$ are not accepting. For a tree t over Σ, instead of considering the accessibility relation $Acc(t)$ as defined in Section 3.1, we consider pairs $((q, q'), r) \in (Q_{\mathcal{A}} \times Q_{\mathcal{A}}) \times 2^{Q_{\mathcal{B}} \times Q_{\mathcal{B}}}$ such that $(q, q') \in Acc_{\mathcal{A}}(t)$ and $r = Acc_{\mathcal{B}}(t)$ for the same tree t. In this way, $t \in L(\mathcal{A}) \setminus L(\mathcal{B})$ iff $(q, q') \in Q_{i,\mathcal{A}} \times Q_{f,\mathcal{A}}$ for some $(q, q') \in Acc_{\mathcal{A}}(t)$, and $Acc_{\mathcal{B}}(t) \cap Q_{i,\mathcal{B}} \times Q_{f,\mathcal{B}} = \emptyset$.

Given a set $\mathscr{S} \subseteq (Q_{\mathcal{A}} \times Q_{\mathcal{A}}) \times 2^{Q_{\mathcal{B}} \times Q_{\mathcal{B}}}$, we define $\mathscr{S}^*$ as the set $\{((p, p'), s) \mid$ there exist $n \geq 0, ((q_i, q'_i), r_i) \in \mathscr{S}$ with $q'_i = q_{i+1}$ for all $1 \leq i < n, p = q_1, p' = q_n,$ and $s = r_1 \circ \cdots \circ r_n\}$. In particular $\mathscr{S}^*$ contains the pairs $((q, q), id_{Q_{\mathcal{B}}})$ for all $q \in Q_{\mathcal{A}}$, obtained when $n = 0$.

The *Post* operator is adapted to $Post_{\subseteq}$ as follows. In this definition, we use notation $Post^{\mathcal{B}}$ to denote the *Post* operator used in automaton $\mathcal{B}$.

Definition 10. *For $\mathscr{S} \subseteq (Q_{\mathcal{A}} \times Q_{\mathcal{A}}) \times 2^{Q_{\mathcal{B}} \times Q_{\mathcal{B}}}$, we define $Post_{\subseteq}(\mathscr{S}) = \mathscr{S}' \cup \mathscr{S}$, where $\mathscr{S}'$ is the set of pairs $((p, p'), s)$ such that*

$$p \xrightarrow{a:\gamma} q \in \Delta_{\mathcal{A}},\ q' \xrightarrow{\overline{a}:\gamma} p' \in \Delta_{\mathcal{A}} \text{ and } s = Post_a^{\mathcal{B}}(r)$$

for some $a \in \Sigma$ and $((q, q'), r) \in \mathscr{S}^*$. Let $Post^*_{\subseteq}(\mathscr{S}) = \cup_{i \geq 0} Post^i_{\subseteq}(\mathscr{S})$ such that $Post^0_{\subseteq}(\mathscr{S}) = \mathscr{S}$, and for all $i > 0$, $Post^i_{\subseteq}(\mathscr{S}) = Post_{\subseteq}(Post^{i-1}_{\subseteq}(\mathscr{S}))$.

Proposition 5 is adapted into the next proposition.

Proposition 11. *Let $\mathcal{A}$ and $\mathcal{B}$ be two VPAs. Then $L(\mathcal{A}) \subseteq L(\mathcal{B})$ iff $\forall((q, q'), r) \in Post^*_{\subseteq}(\emptyset)$, $(q, q') \in Q_{i,\mathcal{A}} \times Q_{f,\mathcal{A}} \Rightarrow r \cap Q_{i,\mathcal{B}} \times Q_{f,\mathcal{B}} \neq \emptyset$.*

An algorithm for testing the inclusion can be designed as done for universality. Again we can save computations by using antichains. In this context, the set $(Q_{\mathcal{A}} \times Q_{\mathcal{A}}) \times 2^{Q_{\mathcal{B}} \times Q_{\mathcal{B}}}$ is equipped with the $\subseteq$ operator such that $((q, q'), r) \subseteq ((p, p'), s)$ iff $(q, q') = (p, p'), r \subseteq s$. As done in Section 3.3, we can define $Post^*_{\subseteq,min}(\emptyset)$ and get the counterpart of Proposition 11 using antichains.

4.2 Universality of General VPAs

We considered VPAs as unranked tree acceptors, i.e. VPAs that only operate on linearizations of trees. We show here how our algorithms can be easily adapted to the original VPA model [2].

Three Types of Symbols - The original VPA model operates on an alphabet partitioned into three disjoint sets: a set Σ_c of call symbols, a set Σ_r of return symbols[3], and a set Σ_i of internal symbols that have no effect on the stack. Such a VPA is called universal if it accepts all words of $(\Sigma_c \cup \Sigma_r \cup \Sigma_i)^*$. Function 1 (resp. Function 3) can be adapted to VPAs with these three types of symbols as follows. We initialize $\mathscr{R}^*$ (resp. $\mathscr{R}^*_{min}$) to $\{id_Q\} \cup \bigcup_{a\in\Sigma_i}\{(q,q') \mid q \xrightarrow{a} q' \in \Delta\}$ instead of $\{id_Q\}$. Indeed, internal symbols can appear at any place in a word, so the relation $\{(q,q') \mid q \xrightarrow{a} q' \in \Delta\}$ has to be combined with any other yielded relation. We also adapt the *Post* operator by defining $Post_{a,\overline{b}}(r) = \{(p,p') \in Q \times Q \mid \exists (q,q') \in r,\ p \xrightarrow{a:\gamma} q \in \Delta,\ q' \xrightarrow{\overline{b}:\gamma} p' \in \Delta\}$ for all $a \in \Sigma_c$ and $\overline{b} \in \Sigma_r$.

Pending Calls and Returns - So far we considered linearizations of trees, but all definitions and results proposed above also hold for linearizations of hedges, formally defined by: $[t_1 \cdots t_n] = [t_1] \cdots [t_n]$. A word w over $\Sigma \cup \overline{\Sigma}$ is not necessarily the linearization of a hedge. The general shape of such a word w is:

$$w = [h_0]\overline{b}_0[h_1]\overline{b}_1 \cdots [h_m]\overline{b}_m[h]a_1[h'_1]a_2[h'_2]\cdots a_n[h'_n]$$

where all h_i, h'_j, and h are hedges of H_Σ, and $\overline{b_i} \in \overline{\Sigma}$, $a_j \in \Sigma$ for all i,j. In other words, w is a sequence of words $[h_i]\overline{b_i}$, followed by the linearization of a hedge $[h]$, followed by a sequence of words $a_j[h'_j]$. The idea here is to adapt Function 1 so that it computes :

- as before, the set $\mathscr{R} = \{Acc(t) \mid t \in T_\Sigma\}$ of all accessibility relations through linearizations of trees, and its closure by composition $\mathscr{R}^*$. This latter set corresponds to the accessibility relation through linearizations of hedges.
- $\mathscr{C}^*$, the closure by composition of $\mathscr{C}$, which is the set of all accessibility relations through the linearization of a hedge, followed by a symbol in $\overline{\Sigma}$: $\mathscr{C} = \{Acc([h]\overline{b}) \mid h \in H_\Sigma,\ \overline{b} \in \overline{\Sigma}\}$.
- $\mathscr{O}^*$, the closure by composition of $\mathscr{O} = \{Acc(a[h]) \mid a \in \Sigma,\ h \in H_\Sigma\}$.

Each time a new relation r is added to $\mathscr{R}$, we update $\mathscr{R}^*$ as previously, using Function CompositionClosure. We also update $\mathscr{C}^*$ by first adding all relations $r \circ r_{\overline{b}}$ (instead of r) where $r_{\overline{b}} = \{(q,q') \mid q \xrightarrow{\overline{b}:\perp} q' \in \Delta\}$, and then compute the closure by composition of $\mathscr{C}^*$ as done for $\mathscr{R}^*$. The same method is used for $\mathscr{O}^*$, with relations $r_a \circ r$. Each time a new relation is added into one of these three sets ($\mathscr{R}^*$, $\mathscr{C}^*$ and $\mathscr{O}^*$), we check whether it is a witness of non-universality. Once these three sets are fully computed, we check whether all relations $r_c \circ r_h \circ r_o$ with $r_c \in \mathscr{C}^*$, $r_h \in \mathscr{R}^*$ and $r_o \in \mathscr{O}^*$ intersect $Q_i \times Q_f$.

4.3 Universality of Hedge Automata

The algorithms presented in this paper are easily adapted from VPAs to hedge automata [5,6]. We provide a translation of hedge automata to VPAs in the long version of the paper. Hence we can derive universality and inclusion algorithms for hedge automata, from the ones we propose for VPAs.

[3] Σ_c and Σ_r are no longer related by the relation $\Sigma_c = \Sigma$ and $\Sigma_r = \overline{\Sigma}$.

5 Experiments

We have implemented the algorithms of Sections 3 and 4 with the described optimizations in a prototype tool named ATC4VPA (AnTiChains for Visibly Pushdown Automata). The code is written mainly in Python, a small part being written in C for the low level operations on arrays of bit-vectors (used to represent relations). The experiments were run on a PC equipped with an Intel i7 2.8GHz processor, 6 GB of RAM and running Linux Ubuntu 3.2.

The experimental tests are performed on randomly generated automata[4], and compared with the results obtained with known prototype tools:

– VPAchecker, a package for deciding universality and inclusion of VPAs, by using either the classical algorithms based on the determinization of VPAs, or optimized on-the-fly algorithms [17],
– FADecider, a package for deciding universality and inclusion of VPAs (over finite and infinite words) using Ramsey-based methods [12],
– OpenNWA, a nested-word automaton library that provides the standard boolean operations [9]. Nested-word automata are another formalism which can be seen as an alternative encoding of VPAs [3].

During the random generation of VPAs, some parameters are fixed: there is one initial state ($|Q_i| = 1$), all states are final ($Q_f = Q$), the alphabets have size 3 ($|\Sigma_c| = |\Sigma_r| = |\Sigma_i| = 3$)[5]. Other parameters vary, like the size $|Q|$, the transition density, i.e. the number of outgoing transitions per state and per alphabet[6], and the number of generated VPAs for a fixed size (sample size).

In Figure 2, we compare ATC4VPA with FADecider for increasing automata sizes, with a fixed transition density, and samples of 100 random automata for each size. The first graph indicates the average time for false (resp. true) instances (without taking into account the timeouts), whereas the second graph indicates the number of false (resp. true) instances that did not reach a timeout fixed to 60 seconds (the number of timeouts is thus the sample size minus these two numbers). By true instances, we mean universal VPAs.

Classical automata techniques (typically complementation) do not scale for universality and inclusion tests. This is the case with OpenNWA, which quickly faces timeouts when automata sizes increase. We now focus on optimized tools.

On all instances, we observe that the *memory footprint* of our tool is much lower than for FADecider. Indeed, antichains reduce the number of relations that have to be computed (and thus stored and processed).

On *true* instances, antichains show their full power, as the state space to be explored is usually huge. For these instances, ATC4VPA is indeed faster than both FADecider and VPAchecker. It also answers to more instances than the

[4] We did not use the same benchmark as [12] as it relies on only few instances, and these VPAs have ϵ-transitions.

[5] $|\Sigma_i| = 0$ when the tested tool offers this possibility.

[6] For instance, a density transition of 5 means 5 outgoing transitions labeled by symbols of Σ_c (Σ_r, Σ_i resp.) for each state.

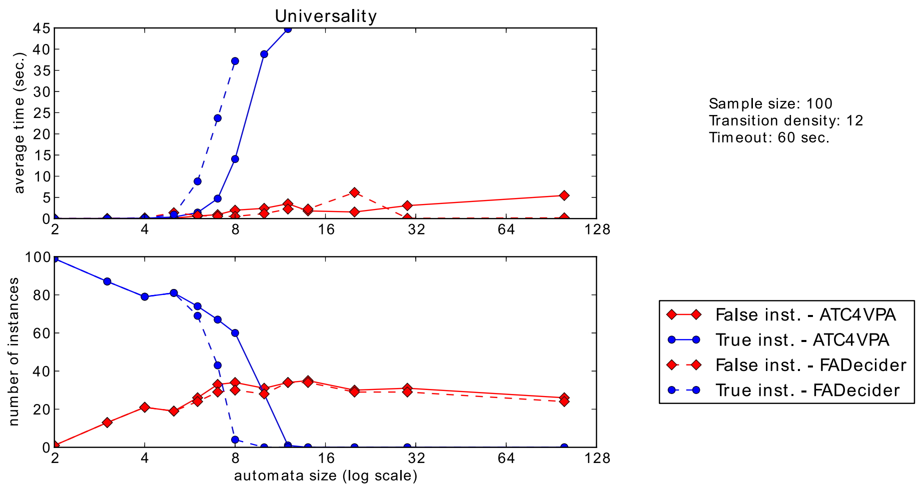


Fig. 2. Universality test for ATC4VPA and FADecider

other tools (less timeouts), the only exception being the universality test in VPAchecker, where VPAchecker encounters slightly less timeouts. Note however that it is difficult to make a fair comparison since VPAchecker outputs a wrong answer for a few true instances.

On *false* instances, our prototype ATC4VPA is a bit slower and faces a bit more timeouts, but with the same asymptotical behavior. This is due to the fact that the data structure for antichains is more involved and our prototype did not optimize all its details.

6 Perspectives

This work suggests future research in several directions. We first plan to investigate another approach, based on the encoding of unranked trees into binary ones, and the algorithm in [4]. Second, we would like to extend our benchmark with realistic (instead of randomly generated) instances coming from the translation of XML schemas or queries to VPAs. Another step is to decide, when a prefix u of a word w has been read, whether w will be accepted by a given VPA (this paper addresses the special case $u = \epsilon$).

Acknowledgements. We thank Thomas Brihaye for useful discussions, and the authors of VPAchecker, FADecider and OpenNWA for helping us using their tools.

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. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: 36th ACM Symposium on Theory of Computing, pp. 202–211. ACM-Press (2004)
3. Alur, R., Madhusudan, P.: Adding nesting structure to words. Journal of the ACM 56(3), 1–43 (2009)
4. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008)
5. Brüggemann-Klein, A., Murata, M., Wood, D.: Regular tree and regular hedge languages over unranked alphabets: Version 1. Tech. Rep. HKTUST-TCSC-2001-05, HKUST Theoretical Computer Science Center Research (2001)
6. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), `http://www.grappa.univ-lille3.fr/tata` (release October 12, 2007)
7. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
8. Doyen, L., Raskin, J.-F.: Antichain Algorithms for Finite Automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 2–22. Springer, Heidelberg (2010)
9. Driscoll, E., Thakur, A., Reps, T.: OpenNWA: A Nested-Word Automaton Library. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 665–671. Springer, Heidelberg (2012)
10. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient Algorithms for Model Checking Pushdown Systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000)
11. Francis, N., David, C., Libkin, L.: A Direct Translation from XPath to Nondeterministic Automata. In: 5th Alberto Mendelzon International Workshop on Foundations of Data Management (2011)
12. Friedmann, O., Klaedtke, F., Lange, M.: Ramsey goes visibly pushdown (2012)
13. Gauwin, O., Niehren, J., Roos, Y.: Streaming tree automata. Information Processing Letters 109(1), 13–17 (2008)
14. Kumar, V., Madhusudan, P., Viswanathan, M.: Visibly pushdown automata for streaming XML. In: 16th International Conference on World Wide Web, pp. 1053–1062. ACM-Press (2007)
15. Löding, C., Lutz, C., Serre, O.: Propositional dynamic logic with recursive programs. The Journal of Logic and Algebraic Programming 73(1-2), 51–69 (2007)
16. Nguyen, T.V.: A tighter bound for the determinization of visibly pushdown automata. In: INFINITY. EPTCS, vol. 10, pp. 62–76 (2009)
17. Van Nguyen, T., Ohsaki, H.: On Model Checking for Visibly Pushdown Automata. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 408–419. Springer, Heidelberg (2012)
18. Srba, J.: Visibly Pushdown Automata: From Language Equivalence to Simulation and Bisimulation. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 89–103. Springer, Heidelberg (2006)

Two-Sided Derivatives for Regular Expressions and for Hairpin Expressions

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

LITIS, Université de Rouen, 76801 Saint-Étienne du Rouvray Cedex, France
{jean-marc.champarnaud,jean-philippe.dubernard}@univ-rouen.fr,
{hadrien.jeanne,ludovic.mignot}@univ-rouen.fr

Abstract. The aim of this paper is to design the polynomial construction of a finite recognizer for hairpin completions of regular languages. This is achieved by considering completions as new expression operators and by applying derivation techniques to the associated extended expressions called hairpin expressions. More precisely, we extend partial derivation of regular expressions to two-sided partial derivation of hairpin expressions and we show how to deduce a recognizer for a hairpin expression from its two-sided derived term automaton, providing an alternative proof of the fact that hairpin completions of regular languages are linear context-free.

Keywords: Finite Automaton, Partial Derivation, Two-sided Derivation, Linear Context-Free Language, Hairpin Completion of Regular Languages, Hairpin Expression.

1 Introduction

The aim of this paper is to design the polynomial construction of a finite recognizer for hairpin completions of regular languages. Given an integer $k > 0$ and an involution H over an alphabet Γ, the hairpin k-completion of two languages L_1 and L_2 over Γ is the language $\mathrm{H}_k(L_1, L_2) = \{\alpha\beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \mid \alpha, \beta, \gamma \in \Gamma^* \wedge (\alpha\beta\gamma\mathrm{H}(\beta) \in L_1 \vee \beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \in L_2) \wedge |\beta| = k\}$ (see Figure 1). The hairpin completion of formal languages has been introduced in [6] by reason of its application to biochemistry. It aroused numerous studies that investigate theoretical and algorithmic properties of hairpin completions or related operations (see for example [8,11,12]). One of the most recent result concerns the problem of deciding regularity of hairpin completions of regular languages; it can be found in [7] as well as a complete bibliography about hairpin completion.

Hairpin completions of regular languages are proved to be linear context-free from [6]. An alternative proof is presented in this paper, with a somehow more constructive approach, since it provides a recognizer for the hairpin completion. This is achieved by considering completions as new expression operators and by applying derivation techniques to the associated extended expressions, that we call hairpin expressions. Notice that a similar derivation-based approach has

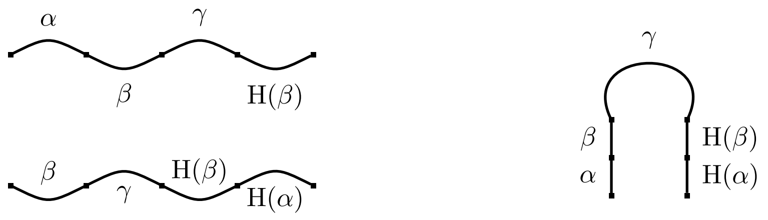


Fig. 1. The Hairpin Completion

been used to study approximate regular expressions [5], through the definition of new distance operators.

Two-sided derivation is shown to be particularly suitable for the study of hairpin expressions. More precisely, we extend partial derivation of regular expressions [1] to two-sided partial derivation of regular expressions first and then of hairpin expressions. We prove that the set of two-sided derived terms of a hairpin expression E over an alphabet Γ is finite. Hence the two-sided derived term automaton A is a finite one. Furthermore the automaton A is over the alphabet $(\Gamma \cup \{\varepsilon\})^2$ and, as we prove it, the language over Γ of such an automaton is linear context-free and not necessarily regular. Finally we show that the language of the hairpin expression E and the language over Γ of the automaton A are equal.

The paper is organized as follows. Next section gathers useful definitions and properties concerning automata and regular expressions. The notion of two-sided residual of a language is introduced in Section 3, as well as the related notion of Γ-couple automaton. In Section 4, hairpin completions of regular languages and their two-sided residuals are investigated. The two-sided partial derivation of hairpin expressions is considered in Section 5, leading to the construction of a finite recognizer.

2 Preliminaries

An *alphabet* is a finite set of distinct symbols. Given an alphabet Σ, we denote by Σ^* the set of all the words over Σ. The empty word is denoted by ε. A *language* over Σ is a subset of Σ^*. The three operations $\cup$, $\cdot$ and * are defined for any two languages L_1 and L_2 over Σ by: $L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \ \vee \ w \in L_2\}$, $L_1 \cdot L_2 = \{w_1 w_2 \in \Sigma^* \mid w_1 \in L_1 \ \wedge \ w_2 \in L_2\}$, $L_1^* = \{\varepsilon\} \cup \{w_1 \cdots w_k \in \Sigma^* \mid \forall j \in \{1, \ldots, k\}, w_j \in L_1\}$. The family of *regular languages* over Σ is the smallest family $\mathcal{F}$ closed under the three operations $\cup$, $\cdot$ and * satisfying $\emptyset \in \mathcal{F}$ and $\forall a \in \Sigma$, $\{a\} \in \mathcal{F}$. Regular languages can be represented by *regular expressions.* A *regular expression* over Σ is inductively defined by: $E = a$, $E = \varepsilon$, $E = \emptyset$, $E = F + G$, $E = F \cdot G$, $E = F^*$, where a is any symbol in Σ and F and G are any two regular expressions over Σ. The *width of* E is the number of occurrences of symbols in E, and its *star number* the number of occurrences of the operator *. The language *denoted by* E is the language $L(E)$ inductively defined by: $L(A) = \{a\}$, $L(\varepsilon) = \{\varepsilon\}$, $L(\emptyset) = \emptyset$, $L(F + G) = L(F) \cup L(G)$, $L(F \cdot G) = L(F) \cdot L(G)$, $L(F^*) = (L(F))^*$, where a is any symbol in Σ and F and G are any two regular expressions over Σ. The language denoted by a regular expression is regular.

Let w be a word in Σ^* and L be a language. The *left residual* (resp. *right residual*) of L w.r.t. w is the language $w^{-1}(L) = \{w' \in \Sigma^* \mid ww' \in L\}$ (resp. $(L)w^{-1} = \{w' \in \Sigma^* \mid w'w \in L\}$). It has been shown that the set of the left residuals (resp. right residuals) of a language is a finite set if and only if the language is regular.

An *automaton* (or a *NFA*) over an alphabet Σ is a 5-tuple $A = (\Sigma, Q, I, F, \delta)$ where Σ is an alphabet, Q a finite set of *states*, $I \subset Q$ the set of *initial states*, $F \subset Q$ the set of *final states* and δ the *transition function* from $Q \times \Sigma$ to 2^Q. The domain of the function δ can be extended to $2^Q \times \Sigma^*$ as follows: for any word w in Σ^*, for any symbol a in Σ, for any set of states $P \subset Q$, for any state $p \in Q$, $\delta(P, \varepsilon) = P$, $\delta(p, aw) = \delta(\delta(p, a), w)$ and $\delta(P, w) = \bigcup_{p\in P} \delta(p, w)$.

The *language recognized* by the automaton A is the set $L(A) = \{w \in \Sigma^* \mid \delta(I, w) \cap F \neq \emptyset\}$. Given a state q in Q, the *right language* of q is the set $\overrightarrow{L}(q) = \{w \in \Sigma^* \mid \delta(q, w) \cap F \neq \emptyset\}$. It can be shown that **(1)** $L(A) = \bigcup_{i\in I} \overrightarrow{L}(i)$, **(2)** $\overrightarrow{L}(q) = \{\varepsilon \mid q \in F\} \cup (\bigcup_{a\in\Sigma, p\in\delta(q,a)} \{a\} \cdot \overrightarrow{L}(p))$ and **(3)** $a^{-1}(\overrightarrow{L}(q)) = \bigcup_{p\in\delta(q,a)} \overrightarrow{L}(p)$.

Kleene Theorem [9] asserts that a language is regular if and only if there exists an NFA that recognizes it. As a consequence, for any language L, there exists a regular expression E such that $L(E) = L$ if and only if there exists an NFA A such that $L(A) = L$. Conversion methods from an NFA to a regular expression and *vice versa* have been deeply studied. In this paper, we focus on the notion of partial derivative defined by Antimirov [1]1.

Given a regular expression E over an alphabet Σ and a word w in Σ^*, the *left partial derivative* of E w.r.t. w is the set $\frac{\partial}{\partial_w}(E)$ of regular expressions satisfying: $\bigcup_{E'\in\frac{\partial}{\partial_w}(E)} L(E') = w^{-1}(L(E))$.

This set is inductively computed as follows: for any two regular expressions F and G, for any word w in Σ^* and for any two distinct symbols a and b in Σ,

$$\frac{\partial}{\partial_a}(a) = \{\varepsilon\},\ \frac{\partial}{\partial_a}(b) = \frac{\partial}{\partial_a}(\varepsilon) = \frac{\partial}{\partial_a}(\emptyset) = \emptyset,$$
$$\frac{\partial}{\partial_a}(F + G) = \frac{\partial}{\partial_a}(F) \cup \frac{\partial}{\partial_a}(G),\ \frac{\partial}{\partial_a}(F^*) = \frac{\partial}{\partial_a}(F) \cdot F^*,$$
$$\frac{\partial}{\partial_a}(F \cdot G) = \begin{cases} \frac{\partial}{\partial_a}(F) \cdot G \cup \frac{\partial}{\partial_a}(G) & \text{if } \varepsilon \in L(F),\\ \frac{\partial}{\partial_a}(F) \cdot G & \text{otherwise,} \end{cases}$$
$$\frac{\partial}{\partial_{aw}}(F) = \frac{\partial}{\partial_w}(\frac{\partial}{\partial_a}(F)),\ \frac{\partial}{\partial_\varepsilon}(F) = \{F\},$$

where for any set $\mathcal{E}$ of regular expressions and for any word w in Σ^*, $\frac{\partial}{\partial_w}(\mathcal{E}) = \bigcup_{E\in\mathcal{E}} \frac{\partial}{\partial_w}(E)$. Any expression appearing in a left partial derivative is called a *left derived term*. Similarly, the *right partial derivative* of a regular expression E over an alphabet Σ w.r.t. a word w in Σ^* is the set $(E)\frac{\partial}{\partial_w}$ inductively defined

1 Partial derivation is investigated in the more general framework of weighted expressions in [10].

as follows for any two regular expressions F and G, for any word w in Σ^* and for any two distinct symbols a and b in Σ,

$$(a)\frac{\partial}{\partial_a} = \{\varepsilon\},\ (b)\frac{\partial}{\partial_a} = (\varepsilon)\frac{\partial}{\partial_a} = (\emptyset)\frac{\partial}{\partial_a} = \emptyset,$$
$$(F+G)\frac{\partial}{\partial_a} = (F)\frac{\partial}{\partial_a} \cup (G)\frac{\partial}{\partial_a},\ (F^*)\frac{\partial}{\partial_a} = F^* \cdot (F)\frac{\partial}{\partial_a},$$
$$(F \cdot G)\frac{\partial}{\partial_a} = \begin{cases} F \cdot (G)\frac{\partial}{\partial_a} \cup (F)\frac{\partial}{\partial_a} & \text{if } \varepsilon \in L(G), \\ F \cdot (G)\frac{\partial}{\partial_a} & \text{otherwise,} \end{cases}$$
$$(F)\frac{\partial}{\partial_{aw}} = ((F)\frac{\partial}{\partial_a})\frac{\partial}{\partial_w},\ (F)\frac{\partial}{\partial_\varepsilon} = \{F\},$$

where for any set $\mathcal{E}$ of regular expressions for any word w in Σ^*, $(\mathcal{E})\frac{\partial}{\partial_w} = \bigcup_{E \in \mathcal{E}} (E)\frac{\partial}{\partial_w}$. Any expression appearing in a right partial derivative is called a *right derived term*. We denote by $\overleftarrow{\mathcal{D}_E}$ (resp. $\overrightarrow{\mathcal{D}_E}$) the set of left (resp. right) derived terms of the expression E. From the set of left derived terms of a regular expression E of width n, Antimirov defined in [1] the *derived term automaton* A of E and showed that A is a n-state NFA that recognizes $L(E)$.

A language over an alphabet Γ is said to be *linear context-free* if it can be generated by a linear grammar, that is a grammar equipped with productions in one of the following forms:

1. $A \to xBy$, where A and B are any two non-terminal symbols, and x and y are any two symbols in $\Gamma \cup \{\varepsilon\}$ such that $(x, y) \neq (\varepsilon, \varepsilon)$,
2. $A \to \varepsilon$, where A is any non-terminal symbol.

Notice that the family of regular languages is strictly included into the family of linear context-free languages. In the following, we will consider combinations of left and right partial derivatives in order to deal with non-regular languages.

3 Two-Sided Residuals of a Language and Couple NFA

In this section, we extend residuals to two-sided residuals. This operation is the composition of left and right residuals, but it is more powerful than classical residuals since it allows to compute a finite subset of the set of residuals even for non-regular languages, which allows to construct a derivative-based finite recognizer.

Definition 1. *Let L be a language over an alphabet Γ and let u and v be two words in Γ^*. The* two-sided residual *of L w.r.t. (u, v) is the language $(u, v)^{-1}(L) = \{w \in \Gamma^* \mid uwv \in L\}$.*

As above-mentioned, the two-sided residual operation is the composition of the two operations of left and right residuals.

Lemma 2. *Let L be a language over an alphabet Γ and u and v be two words in Γ^*. Then: $(u, v)^{-1}(L) = (u^{-1}(L))v^{-1} = u^{-1}((L)v^{-1})$.*

Corollary 3. *Let L be a language over an alphabet Γ and u and v be two words in Γ^*. Then: $\varepsilon \in (u, v)^{-1}(L) \Leftrightarrow uv \in L$.*

It is a folk knowledge that NFAs are related to left residual computation according to the following assertion **(A)**: in an NFA $(\Sigma, Q, I, F, \delta)$, a word aw belongs to $\overrightarrow{L}(q)$ with $q \in Q$ if and only if w belongs to $a^{-1}(\overrightarrow{L}(q)) = \bigcup_{q' \in \delta(q,a)} \overrightarrow{L}(q')$. Since a two-sided residual w.r.t. a couple (x, y) of symbols in an alphabet Γ is by definition the combination of a left residual w.r.t. x and of a right residual w.r.t. y, the assertion **(A)** can be extended to two-sided residuals by introducing *couple NFAs* equipped with transitions labelled by couples of symbols in Γ. The notion of right language of a state is extended to the one of Γ-right language as follows: if a given word w in Γ^* belongs to the Γ-right language of a state q' and if there exists a transition from a state q to q' labelled by a couple (x, y), then the word xwy belongs to the Γ-right language of q.

More precisely, given an alphabet Γ, we set $\Sigma_\Gamma = \{(x, y) \mid x, y \in \Gamma \cup \{\varepsilon\} \wedge (x, y) \neq (\varepsilon, \varepsilon)\}$. We consider the mapping Im from $(\Sigma_\Gamma)^*$ to Γ^* inductively defined for any word w in $(\Sigma_\Gamma)^*$ and for any symbol $(x, y) \in \Sigma_\Gamma$ by: $\mathrm{Im}(\varepsilon) = \varepsilon$ and $\mathrm{Im}((x, y) \cdot w) = x \cdot \mathrm{Im}(w) \cdot y$. Notice that this mapping was introduced by Sempere [13] in order to compute the language denoted by a *linear expression.* Linear expressions denote linear context-free languages, and are equivalent to the *regular-like expressions* of Brzozowski [2].

Definition 4. *Let $A = (\Sigma, Q, I, F, \delta)$ be an NFA. The NFA A is a* couple NFA *if there exists an alphabet Γ such that $\Sigma \subset \Sigma_\Gamma$. In this case, A is called a Γ-*couple NFA*. The Γ-*language *of a Γ-couple NFA A is the subset $L_\Gamma(A)$ of Γ^* defined by: $L_\Gamma(A) = \{\mathrm{Im}(w) \mid w \in L(A)\}$.*

The definition of right languages and their classical properties extend to couple NFAs as follows. Let $A = (\Sigma, Q, I, F, \delta)$ be a Γ-couple NFA and q be a state in Q. The *Γ-right language* of q is the subset $\overrightarrow{L}_\Gamma(q)$ of Γ^* defined by: $\overrightarrow{L}_\Gamma(q) = \{\mathrm{Im}(w) \mid w \in \overrightarrow{L}(q)\}$.

Lemma 5. *Let $A = (\Sigma, Q, I, F, \delta)$ be a Γ-couple NFA and q be a state in Q. Then: $L_\Gamma(A) = \bigcup_{i \in I} \overrightarrow{L}_\Gamma(i)$.*

Lemma 6. *Let $A = (\Sigma, Q, I, F, \delta)$ be a Γ-couple NFA and q be a state in Q. Then: $\overrightarrow{L}_\Gamma(q) = \{\varepsilon \mid q \in F\} \cup \bigcup_{(x,y) \in \Sigma, q' \in \delta(q,(x,y))} \{x\} \cdot \overrightarrow{L}_\Gamma(q') \cdot \{y\}$.*

Corollary 7. *Let $A = (\Sigma, Q, I, F, \delta)$ be a Γ-couple NFA, (x, y) be a couple in Σ_Γ and q be a state in Q. Then: $(x, y)^{-1}(\overrightarrow{L}_\Gamma(q)) = \bigcup_{q' \in \delta(q,(x,y))} \overrightarrow{L}_\Gamma(q')$.*

The following example illustrates the fact that there exist non-regular languages that can be recognized by couple NFAs.

Example 8. Let $\Gamma = \{a, b\}$ and A be the automaton of the Figure 2. The Γ-language of A is $L_\Gamma(A) = \{a^n b^n \mid n \in \mathbb{N}\}$.

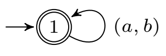


Fig. 2. The Couple Automaton A

As a consequence there exist context free languages that are recognized by a couple NFA. In fact, the family of languages recognized by couple NFAs is exactly the family of linear context-free languages.

Theorem 9. *A language is linear context-free if and only if it is recognized by a couple NFA.*

We present here two algorithms in order to solve the membership problem[2] *via* a couple NFA. The Algorithm 2 checks whether the word $w \in \Gamma^*$ is recognized by the Γ-couple NFA A. It returns TRUE if there exists an initial state such that its Γ-right language contains w. The Algorithm 1 checks whether the word $w \in \Gamma^*$ is in the Γ-right language of the state q.

```
begin
    if $w = \varepsilon$ then
        $P \leftarrow (q \in F)$;
    end
    else
        $P \leftarrow$ FALSE;
        foreach $(q, (\alpha, \beta), q') \in \delta \mid w = \alpha w' \beta$ do
            $P \leftarrow P \vee$ RightLanguage($A$, $w'$, $q'$);
        end
    end
    return $P$
end
```

Algorithm 1. RightLanguage(A,w,q)

```
begin
    $R \leftarrow$ FALSE;
    foreach $i \in I$ do
        $R \leftarrow R \vee$ RightLanguage($A$, $w$, $i$);
    end
    return $R$
end
```

Algorithm 2. MembershipTest(A,w)

Proposition 10. *Let $A = (\Sigma, Q, I, F, \delta)$ be a Γ-couple NFA, q be a state in Q and w be a word in Γ^*. The two following propositions are satisfied:*

1. *Algorithm 1: RightLanguage(A, w, q) returns* $(w \in \overrightarrow{L}_\Gamma(q))$,
2. *Algorithm 2: MembershipTest(A,w) returns* $(w \in L_\Gamma(A))$.

The following sections are devoted to hairpin completions and their two-sided residuals. It turns out that hairpin completions are linear context-free. Hence, we show how to compute a couple NFA that recognizes a given hairpin completion.

[2] Given a language L and a word w, does w belong to L?

4 Hairpin Completion of a Language and Its Residuals

Let Γ be an alphabet. An *involution* f over Γ is a mapping from Γ to Γ satisfying for any symbol a in Γ, $\mathrm{f}(\mathrm{f}(a)) = a$. An *anti-morphism* μ over Γ^* is a mapping from Γ^* to Γ^* satisfying for any two words u and v in Γ^* $\mu(u \cdot v) = \mu(v) \cdot \mu(u)$. Any mapping g from Γ to Γ can be extended as an anti-morphism over Γ^* as follows: $\forall a \in \Gamma$, $\forall w \in \Gamma^*$, $\mathrm{g}(\varepsilon) = \varepsilon$, $\mathrm{g}(a \cdot w) = \mathrm{g}(w) \cdot \mathrm{g}(a)$.

Definition 11. *Let Γ be an alphabet and* H *be an anti-morphism over Γ^*. Let L_1 and L_2 be two languages over Γ. Let $k > 0$ be an integer. The (H, k)-*completion *of L_1 and L_2 is the language* $\mathrm{H}_k(L_1, L_2)$ *defined by:*

$$\mathrm{H}_k(L_1, L_2) = \{\alpha\beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \mid \alpha, \beta, \gamma \in \Gamma^* \wedge (\alpha\beta\gamma\mathrm{H}(\beta) \in L_1 \vee \beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \in L_2) \wedge |\beta| = k\}.$$

The (H, k)-completion operator can be defined as the union of two unary operators $\overleftarrow{\mathrm{H}_k}$ and $\overrightarrow{\mathrm{H}_k}$.

Definition 12. *Let Γ be an alphabet and* H *be an anti-morphism over Γ^*. Let L be a language over Γ. Let $k > 0$ be an integer. The right (resp. left) (H, k)-*completion *of L is the language $\overrightarrow{\mathrm{H}_k}(L)$ (resp. $\overleftarrow{\mathrm{H}_k}(L)$) defined by:*

$$\overrightarrow{\mathrm{H}_k}(L) = \{\alpha\beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \mid \alpha, \beta, \gamma \in \Gamma^* \ \wedge \ \alpha\beta\gamma\mathrm{H}(\beta) \in L \wedge |\beta| = k\},$$
$$\overleftarrow{\mathrm{H}_k}(L) = \{\alpha\beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \mid \alpha, \beta, \gamma \in \Gamma^* \ \wedge \ \beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \in L \wedge |\beta| = k\}.$$

Lemma 13. *Let Γ be an alphabet and* H *be an anti-morphism over Γ^*. Let L_1 and L_2 be two languages over Γ. Let $k > 0$ be an integer. Then:*

$$\mathrm{H}_k(L_1, L_2) = \overrightarrow{\mathrm{H}_k}(L_1) \cup \overleftarrow{\mathrm{H}_k}(L_2).$$

When H is an involution over Γ, the (H, k)-completion of L_1 and L_2 is called a hairpin completion [6]. Even in the case where H is not an involution, we will say that languages such as $\overrightarrow{\mathrm{H}_k}(L)$, $\overleftarrow{\mathrm{H}_k}(L)$ or $\mathrm{H}_k(L, L')$ are hairpin completed languages and we will speak of hairpin completions. We first establish formulae in this general setting in order to compute the two-sided residuals of the completed language of an arbitrary language. The following operator is useful.

Definition 14. *Let Γ be an alphabet and* H *be an anti-morphism over Γ^*. Let L be a language over an alphabet Γ. Let $k > 0$ be an integer. The language $\mathrm{H}'_k(L)$ is defined by: $\mathrm{H}'_k(L) = \{\beta\gamma\mathrm{H}(\beta) \in L \mid \beta, \gamma \in \Gamma^* \ \wedge \ |\beta| = k\}$.*

We split the computation of two-sided residuals of a completed language w.r.t. (x, y) couples: the first case is when both x and y are symbols.

Proposition 15. *Let Γ be an alphabet and* H *be an anti-morphism over Γ^*. Let L be a language over Γ. Let (x, y) a couple of symbols in $\Gamma \times \Gamma$. Let $k > 0$ be an integer. Then:*

$$(x,y)^{-1}(\overrightarrow{\mathrm{H}_k}(L)) = \begin{cases} \emptyset & \textit{if } y \neq \mathrm{H}(x), \\ \overrightarrow{\mathrm{H}_k}(x^{-1}(L)) \cup (x,y)^{-1}(L) & \textit{if } y = \mathrm{H}(x) \ \wedge \ k = 1, \\ \overrightarrow{\mathrm{H}_k}(x^{-1}(L)) \cup \mathrm{H}'_{k-1}((x,y)^{-1}(L)) & \textit{otherwise,} \end{cases}$$
$$(x,y)^{-1}(\overleftarrow{\mathrm{H}_k}(L)) = \begin{cases} \emptyset & \textit{if } y \neq \mathrm{H}(x), \\ \overleftarrow{\mathrm{H}_k}((L)y^{-1}) \cup (x,y)^{-1}(L) & \textit{if } y = \mathrm{H}(x) \ \wedge \ k = 1, \\ \overleftarrow{\mathrm{H}_k}((L)y^{-1}) \cup \mathrm{H}'_{k-1}((x,y)^{-1}(L)) & \textit{otherwise,} \end{cases}$$
$$(x,y)^{-1}(\mathrm{H}'_k(L)) = \begin{cases} \emptyset & \textit{if } y \neq \mathrm{H}(x), \\ \mathrm{H}'_{k-1}((x,y)^{-1}(L)) & \textit{if } y = \mathrm{H}(x) \ \wedge \ k > 1, \\ (x,y)^{-1}(L) & \textit{otherwise.} \end{cases}$$

The problem of two-sided residuals of a completed language w.r.t. couples (x, y) with either x or y equal to ε is that they add one catenation that has to be memorized. It can be checked that this may lead to infinite sets of two-sided residuals.

Proposition 16. *Let Γ be an alphabet and* H *be an anti-morphism over Γ^*. Let L be a language over an alphabet Γ. Let $k > 0$ be an integer. Let L' be a language in $\{\overleftarrow{\mathrm{H}_k}(L), \overrightarrow{\mathrm{H}_k}(L), \mathrm{H}'_k(L)\}$. Let x be a symbol in Γ. Then:*

$$(x,\varepsilon)^{-1}(L') = (x,\mathrm{H}(x))^{-1}(L') \cdot \{\mathrm{H}(x)\},$$
$$(\varepsilon,x)^{-1}(L') = \bigcup_{z \in \Gamma \mid \mathrm{H}(z)=x} \{z\} \cdot (z,x)^{-1}(L').$$

Let L be a language over an alphabet Γ. The set $\mathcal{R}_L$ *of two-sided residuals* of L is defined by: $\mathcal{R}_L = \bigcup_{k \geq 1} \mathcal{R}_L^k$, where

$$\mathcal{R}_L^k = \begin{cases} \{(x,y)^{-1}(L) \mid (x,y) \in \Sigma_\Gamma\} & \text{if } k = 1, \\ \{(x,y)^{-1}(L') \mid (x,y) \in \Sigma_\Gamma \wedge \ L' \in \mathcal{R}_L^{k-1}\} & \text{otherwise.} \end{cases}$$

From now on we focus on hairpin completion of regular languages. Let us recall that such a completion is not necessarily regular [6].

Lemma 17. *The family of regular languages is not closed under hairpin completion.*

Proof. Let $\Gamma = \{a, b, c\}$, $k > 0$ be a fixed integer and H be the anti-morphism over Γ^* defined by $\mathrm{H}(a) = a$, $\mathrm{H}(b) = c$ and $\mathrm{H}(c) = b$. Let $L' = \overrightarrow{\mathrm{H}_k}(L(a^* b^k c^k))$. Let us first show that $L' = \{a^n b^k c^k a^n \mid n \geq 0\}$. Let w be a word in Γ^*.

$$\begin{aligned} w \in L' &\Leftrightarrow w = \alpha\beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \wedge \alpha\beta\gamma\mathrm{H}(\beta) \in L(a^* b^k c^k) \wedge |\beta| = k \\ &\Leftrightarrow w = \alpha\beta\gamma\mathrm{H}(\beta)\mathrm{H}(\alpha) \wedge \alpha \in L(a^*) \wedge \mathrm{H}(\beta) = c^k \wedge \beta = b^k \\ &\Leftrightarrow w = a^n b^k c^k a^n \text{ with } n \geq 0. \end{aligned}$$

For any integer $j \geq 0$, let us define the language L'_j by:

$$L'_j = \begin{cases} L' & \text{if } j = 0, \\ a^{-1}(L'_{j-1}) & \text{otherwise.} \end{cases}$$

Consequently, it holds $L'_j = \{a^{n-j} b^k c^k a^n \mid n \geq j\}$. Finally, since for any two distinct integers j and j', the word $b^k c^k a^j$ belongs to $L'_j \setminus L'_{j'}$, it holds that for any two distinct integers j and j', $L'_j \neq L'_{j'}$ and $(a^j)^{-1}(L') \neq (a^{j'})^{-1}(L')$. As a consequence, the set of left residuals of L' is infinite. □

The set of two-sided residuals of a hairpin completion of a regular language may be infinite, but the restriction to residuals w.r.t. couples (x, y) of symbols is sufficient to obtain a finite set of two-sided residuals and a finite recognizer.

5 The Two-Sided Derived Term Automaton

The computation of residuals is intractable when it is defined over languages. However, derived terms of regular expressions denote residuals of regular languages. We then extend the partial derivation of regular expressions [1] to the partial derivation of hairpin expressions.

A *hairpin expression* E over an alphabet Γ is a regular expression over Γ or is inductively defined by: $E = \overleftarrow{\mathrm{H}_k}(F)$, $E = \overrightarrow{\mathrm{H}_k}(F)$, $E = \mathrm{H}'_k(F)$, $E = G_1 + G_2$, where H is any anti-morphism over Γ^*, $k > 0$ is any integer, F is any regular expression over Γ, and G_1 and G_2 are any two hairpin expressions over Σ. If the only operators appearing in E are regular operators ($+$, $\cdot$ or *), the expression E is said to be a *simple hairpin expression.* The *language* denoted by a hairpin expression E over an alphabet Γ is the regular language $L(E)$ if E is a regular expression or is inductively defined by: $L(\overleftarrow{\mathrm{H}_k}(F)) = \overleftarrow{\mathrm{H}_k}(L(F))$, $L(\overrightarrow{\mathrm{H}_k}(F)) = \overrightarrow{\mathrm{H}_k}(L(F))$, $L(\mathrm{H}'_k(F)) = \mathrm{H}'_k(L(F))$, $L(G_1 + G_2) = L(G_1) \cup L(G_2)$, where H is any anti-morphism over Γ^*, $k > 0$ is any integer, F is any regular expression over Γ, and G_1 and G_2 are any two hairpin expressions over Γ.

Definition 18. *Let E be a hairpin expression over an alphabet Γ. Let (x, y) be a couple of symbols in Σ_Γ. Let $k > 0$ be an integer. The* two-sided partial derivative *of E* w.r.t. (x, y) *is the set $\frac{\partial}{\partial_{(x,y)}}(E)$ of hairpin expressions defined by:*

$$\frac{\partial}{\partial_{(x,y)}}(F) = \begin{cases} (F)\frac{\partial}{\partial_y} & \textit{if } x = \varepsilon, \\ \frac{\partial}{\partial_x}(F) & \textit{if } y = \varepsilon, \\ \bigcup_{F' \in \frac{\partial}{\partial_x}(F)} (F')\frac{\partial}{\partial_y} & \textit{otherwise,} \end{cases}$$

$$\frac{\partial}{\partial_{(x,y)}}(\overrightarrow{\mathrm{H}_k}(F)) = \begin{cases} \emptyset & \textit{if } y \neq \mathrm{H}(x), \\ \overrightarrow{\mathrm{H}_k}(\frac{\partial}{\partial_x}(F)) \cup \frac{\partial}{\partial_{(x,y)}}(F) & \textit{if } y = \mathrm{H}(x) \ \wedge \ k = 1 \\ \overrightarrow{\mathrm{H}_k}(\frac{\partial}{\partial_x}(F)) \cup \mathrm{H}'_{k-1}(\frac{\partial}{\partial_{(x,y)}}(F)) & \textit{otherwise,} \end{cases}$$

$$\frac{\partial}{\partial_{(x,y)}}(\overleftarrow{\mathrm{H}_k}(F)) = \begin{cases} \emptyset & \textit{if } y \neq \mathrm{H}(x), \\ \overleftarrow{\mathrm{H}_k}((F)\frac{\partial}{\partial_y}) \cup \frac{\partial}{\partial_{(x,y)}}(F) & \textit{if } y = \mathrm{H}(x) \ \wedge \ k = 1 \\ \overleftarrow{\mathrm{H}_k}((F)\frac{\partial}{\partial_y}) \cup \mathrm{H}'_{k-1}(\frac{\partial}{\partial_{(x,y)}}(F)) & \textit{otherwise,} \end{cases}$$

$$\frac{\partial}{\partial_{(x,y)}}(\mathrm{H}'_k(F)) = \begin{cases} \emptyset & \textit{if } y \neq \mathrm{H}(x), \\ \mathrm{H}'_{k-1}(\frac{\partial}{\partial_{(x,y)}}(F)) & \textit{if } k¿1, \\ \frac{\partial}{\partial_{(x,y)}}(F) & \textit{otherwise,} \end{cases}$$

$$\frac{\partial}{\partial_{(x,y)}}(G_1 + G_2) = \frac{\partial}{\partial_{(x,y)}}(G_1) \cup \frac{\partial}{\partial_{(x,y)}}(G_2),$$

where H *is any anti-morphism over Γ^*, $k > 0$ is any integer, F is any regular expression over Γ, G_1 and G_2 are any two hairpin expressions over Γ, and for any set $\mathcal{H}$ of hairpin expressions: $\overrightarrow{\mathrm{H}_k}(\mathcal{H}) = \{\overrightarrow{\mathrm{H}_k}(H) \mid H \in \mathcal{H}\}$, $\overleftarrow{\mathrm{H}_k}(\mathcal{H}) = \{\overleftarrow{\mathrm{H}_k}(H) \mid H \in \mathcal{H}\}$, $\mathrm{H}'_k(\mathcal{H}) = \{\mathrm{H}'_k(H) \mid H \in \mathcal{H}\}$.*

Let E be a hairpin expression over an alphabet Γ. The *set* $\overleftrightarrow{\mathcal{D}_E}$ *of two-sided derived terms of the expression* E is defined by: $\overleftrightarrow{\mathcal{D}_E} = \bigcup_{k\geq 1} \overleftrightarrow{\mathcal{D}^k_E}$, where:

$$\overleftrightarrow{\mathcal{D}^k_E} = \begin{cases} \bigcup_{(x,y)\in\Sigma_\Gamma} \frac{\partial}{\partial(x,y)}(E) & \text{if } k=1,\\ \bigcup_{(x,y)\in\Sigma_\Gamma, E'\in\overleftrightarrow{\mathcal{D}^{k-1}_E}} \frac{\partial}{\partial(x,y)}(E') & \text{otherwise.}\end{cases}$$

Derived terms of regular expressions are related to left residuals. Let us show that derived terms of hairpin expressions are related to two-sided residuals.

Proposition 19. *Let E be a hairpin expression over an alphabet Γ. Let (x,y) be a couple of symbols in Γ^2. Then: $\bigcup_{F\in\frac{\partial}{\partial(x,y)}(E)} L(F) = (x,y)^{-1}(L(E))$.*

Furthermore, if E is a regular expression, the proposition still holds whenever (x,y) is a couple of symbols in Σ_Γ.

Determining whether the empty word belongs to the language denoted by a regular expression E can be performed syntactically and inductively as follows: $\varepsilon\notin L(a)$, $\varepsilon\notin L(\emptyset)$, $\varepsilon\in L(\varepsilon)$, $\varepsilon\in L(G_1\cdot G_2)\Leftrightarrow\varepsilon\in L(G_1)\ \wedge\ \varepsilon\in L(G_2)$, $\varepsilon\in L(G_1+G_2)\Leftrightarrow\varepsilon\in L(G_1)\ \vee\ \varepsilon\in L(G_2)$, $\varepsilon\in L(G_1^*)$.

This syntactical test is needed to compute the derived term automaton since it defines the finality of the states. We now show how to extend this computation to hairpin expressions.

Lemma 20. *Let F be a regular expression and G_1 and G_2 be two hairpin expressions. Then: $\varepsilon\notin L(\overrightarrow{\mathrm{H}_k}(F))$, $\varepsilon\notin L(\overleftarrow{\mathrm{H}_k}(F))$, $\varepsilon\notin L(\mathrm{H}'_k(F))$, $\varepsilon\in L(G_1+G_2)\Leftrightarrow\varepsilon\in L(G_1)\ \vee\ \varepsilon\in L(G_2)$.*

The following example illustrates the computation of derived terms. For clarity, in this example, we assume that hairpin expressions are quotiented w.r.t. the following rules: $\varepsilon\cdot E\sim E$, $\emptyset\cdot E\sim\emptyset$. Moreover, sets of expressions are also quotiented w.r.t. the following rule: $\{\emptyset\}\sim\emptyset$.

Example 21. Let $\Gamma=\{a,b,c\}$ and H be the anti-morphism over Γ^* defined by $\mathrm{H}(a)=a$, $\mathrm{H}(b)=c$ and $\mathrm{H}(c)=b$. Let $E=\overrightarrow{\mathrm{H}_1}(a^*bc)$. Derived terms of E are computed as follows: $\frac{\partial}{\partial(a,a)}(E)=\{E\}$, $\frac{\partial}{\partial(b,c)}(E)=\{\overrightarrow{\mathrm{H}_1}(c),\varepsilon\}$, $\frac{\partial}{\partial(c,b)}(\overrightarrow{\mathrm{H}_1}(c))=\{\overrightarrow{\mathrm{H}_1}(\varepsilon)\}$. Other partial derivatives are equal to $\emptyset$. Furthermore, it holds that ε is the only derived term F of E such that ε belongs to $L(F)$.

In the following we are looking for an upper bound for the cardinality of the set of two-sided derived terms, thus we apply no reduction to the regular expressions. Notice that this cardinality decreases whenever any reduction is applied.

Proposition 22. *Let E be a regular expression of width $n>0$ and of star number h. Let us set $m=n+h$. Then the three following propositions hold:*

1. $\mathrm{Card}(\overleftarrow{\mathcal{D}_E})\leq n$,
2. $\mathrm{Card}(\overrightarrow{\mathcal{D}_E})\leq n$,
3. $\mathrm{Card}(\overleftrightarrow{\mathcal{D}_E})\leq\frac{2m\times(m+1)\times(m+2)}{3}-3$.

Proposition 23. *Let E be a regular expression over an alphabet Γ,* H *be an antimorphism over Γ^* and $k > 0$ be an integer. Then:*

1. $\mathrm{Card}(\overleftrightarrow{\mathcal{D}_{\mathrm{H}'_k(E)}}) \leq k \times \mathrm{Card}(\overleftrightarrow{\mathcal{D}_E})$,
2. $\mathrm{Card}(\overleftrightarrow{\mathcal{D}_{\overrightarrow{\mathrm{H}_k}(E)}}) \leq \mathrm{Card}(\overleftarrow{\mathcal{D}_E}) + k \times \mathrm{Card}(\overleftrightarrow{\mathcal{D}_E})$,
3. $\mathrm{Card}(\overleftrightarrow{\mathcal{D}_{\overleftarrow{\mathrm{H}_k}(E)}}) \leq \mathrm{Card}(\overrightarrow{\mathcal{D}_E}) + k \times \mathrm{Card}(\overleftrightarrow{\mathcal{D}_E})$.

The *index* of a hairpin expression E is the integer $\mathrm{index}(E)$ inductively defined by: $\mathrm{index}(F) = 0$, $\mathrm{index}(\overleftarrow{\mathrm{H}_k}(F)) = k$, $\mathrm{index}(\overrightarrow{\mathrm{H}_k}(F)) = k$, $\mathrm{index}(\mathrm{H}'_k(F)) = k$, $\mathrm{index}(G_1 + G_2) = \max(\mathrm{index}(G_1), \mathrm{index}(G_2))$, where H is any anti-morphism over Γ^*, $k > 0$ is any integer, F is any regular expression over Γ, and G_1 and G_2 are any two hairpin expressions over Γ.

Proposition 24. *Let E be a hairpin expression over an alphabet Γ. Then $\overleftrightarrow{\mathcal{D}_E}$ is a finite set the cardinal of which is upper bounded by $k \times (\frac{2m(m+1)(m+2)}{3} - 3) + n$, where k is the index of E, and $m = n + h$ with n its width and h its star number.*

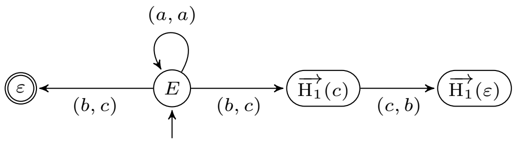
This finite set of two-sided derived terms allows us to extend the finite derived term automaton to hairpin expressions.

Definition 25. *Let E be a hairpin expression over an alphabet Γ. Let $A = (\Sigma_\Gamma, Q, I, F, \delta)$ be the NFA defined by: $Q = \{E\} \cup \overleftrightarrow{\mathcal{D}_E}$, $I = \{E\}$, $F = \{E' \in Q \mid \varepsilon \in L(E')\}$, $\forall (x,y) \in \Sigma_\Gamma, \forall E' \in Q$, $\delta(E', (x,y)) = \frac{\partial}{\partial_{(x,y)}}(E')$.*

The automaton A is the two-sided derived term automaton *of E.*

By construction, A is a Γ-couple NFA where Γ is the alphabet of E.

Example 26. Let E be the hairpin expression of Example 21. The derived term automaton of E is the automaton presented in Figure 3.



$$\begin{tikzcd} \varepsilon & E \arrow[l, "(b,c)"'] \arrow[r, "(b,c)"'] \arrow[loop above, "(a,a)"] & \overrightarrow{\mathrm{H}_1}(c) \arrow[r, "(c,b)"'] & \overrightarrow{\mathrm{H}_1}(\varepsilon) \end{tikzcd}$$

Fig. 3. The Derived Term Automaton of the Expression E

Theorem 27. *Let A be the two-sided derived term automaton of a hairpin expression E over an alphabet Γ and let k be the index of E. Then $L_\Gamma(A) = L(E)$. Furthermore A has at most $k \times (\frac{2m\times(m+1)\times(m+2)}{3} - 3) + n + 1$ states where $m = n + h$, with n the width of E and h its star number.*

Finally, the computation of the two-sided derived term automaton provides an alternative proof of the following theorem.

Theorem 28. *The language denoted by a hairpin expression is linear context-free.*

6 Conclusion

This paper provides an alternative proof of the fact that hairpin completions of regular languages are linear context-free. This proof is obtained by considering the family of regular expressions extended to hairpin operators and by computing their partial derivatives, a technique that has already been applied to regular expressions extended to boolean operators [3], to multi-tilde-bar operators [4] and to approximate operators [5]. Moreover it is a constructive proof since it is based on the computation of a polynomial size recognizer for hairpin completions of regular languages. Let us add that it is possible to compute a linear size recognizer for $(H, 0)$-completions of regular languages.

References

1. Antimirov, V.: Partial derivatives of regular expressions and finite automaton constructions. Theoret. Comput. Sci. 155, 291–319 (1996)
2. Brzozowski, J.A.: Regular-like expressions for some irregular languages. In: SWAT (FOCS), pp. 278–286. IEEE Computer Society (1968)
3. Caron, P., Champarnaud, J.-M., Mignot, L.: Partial Derivatives of an Extended Regular Expression. In: Dediu, A.-H., Inenaga, S., Martín-Vide, C. (eds.) LATA 2011. LNCS, vol. 6638, pp. 179–191. Springer, Heidelberg (2011)
4. Caron, P., Champarnaud, J.-M., Mignot, L.: Multi-Tilde-Bar Derivatives. In: Moreira, N., Reis, R. (eds.) CIAA 2012. LNCS, vol. 7381, pp. 321–328. Springer, Heidelberg (2012)
5. Champarnaud, J.-M., Jeanne, H., Mignot, L.: Approximate Regular Expressions and Their Derivatives. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 179–191. Springer, Heidelberg (2012)
6. Cheptea, D., Martìn-Vide, C., Mitrana, V.: A new operation on words suggested by DNA biochemistry: hairpin completion. Transgressive Computing, 216–228 (2006)
7. Diekert, V., Kopecki, S., Mitrana, V.: Deciding regularity of hairpin completions of regular languages in polynomial time. Inf. Comput. 217, 12–30 (2012)
8. Kari, L., Seki, S., Kopecki, S.: On the regularity of iterated hairpin completion of a single word. Fundam. Inform. 110(1-4), 201–215 (2011)
9. Kleene, S.: Representation of events in nerve nets and finite automata. Automata Studies Ann. Math. Studies 34, 3–41 (1956)
10. Lombardy, S., Sakarovitch, J.: Derivatives of rational expressions with multiplicity. Theor. Comput. Sci. 332(1-3), 141–177 (2005)
11. Manea, F., Martín-Vide, C., Mitrana, V.: On some algorithmic problems regarding the hairpin completion. Discrete Applied Mathematics 157(9), 2143–2152 (2009)
12. 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)
13. Sempere, J.M.: On a class of regular-like expressions for linear languages. Journal of Automata, Languages and Combinatorics 5(3), 343–354 (2000)

How to Travel between Languages[*]

Krishnendu Chatterjee[1], Siddhesh Chaubal[2], and Sasha Rubin[1,3]

[1] IST Austria, Am Campus 1, 3400 Klosterneuburg, Austria
{Krishnendu.Chatterjee,sasha.rubin}@ist.ac.at
[2] IIT Bombay, Powai, Mumbai 400076, India
spchaubal@gmail.com
[3] TU Wien, Institut für Informationssysteme 184/4
Favoritenstraße 911, 1040, Vienna, Austria

Abstract. We consider how to edit strings from a source language so that the edited strings belong to a target language, where the languages are given as deterministic finite automata. Non-streaming (or offline) transducers perform edits given the whole source string. We show that the class of deterministic one-pass transducers with registers along with increment and min operation suffices for computing optimal edit distance, whereas the same class of transducers without the min operation is not sufficient. Streaming (or online) transducers perform edits as the letters of the source string are received. We present a polynomial time algorithm for the *partial-repair problem* that given a bound α asks for the construction of a deterministic streaming transducer (if one exists) that ensures that the 'maximum fraction' η of the strings of the source language are edited, within cost α, to the target language.

Keywords: Edit-distance, Markov decision process, register automaton.

1 Introduction

One of the classic problems in language theory concerns optimally editing an input string so that it belongs to a given target regular language [3].

Definition 1 (Edit-distance). *An* edit operation *applied to a string u either deletes a single character, inserts a single character, or changes a single character. The* edit-distance *between $u, v \in \Sigma^*$, denoted $\mathrm{ED}(u, v)$, is defined as the length of a shortest sequence of edit operations that applied to u yields v. For language T, we define $\mathrm{ED}(u, T) := \inf_{v \in T} \mathrm{ED}(u, v)$.*

In [2] the problem was generalized: given source language R and target language T, how to edit strings from R so that the edited strings belong to T.

[*] The research was supported by Austrian Science Fund (FWF) Grant No P 23499-N23, FWF NFN Grant No S11407-N23 (RiSE), ERC Start grant (279307: Graph Games), and Microsoft faculty fellows award. Thanks to Gabriele Puppis for suggesting the problem of identifying a deterministic transducer to compute the optimal cost, and to Martin Chmelik for his comments on the introduction.

There are two broad categories of transducers $\mathfrak{Tr}$ that edit strings depending on how they process input. The *non-streaming (or offline)* transducers reads the complete source string $u \in R$ and then output the repaired string $\mathfrak{Tr}(u) \in T$. The *streaming (or online)* transducers perform the edits as the letters of input u are received. For a class of transducers, the main interest is to compare $u \mapsto \textsc{ed}(u, \mathfrak{Tr}(u))$ — ie. the cost of editing using $\mathfrak{Tr}$ from the class — to $u \mapsto \textsc{ed}(u, T)$, the cost of optimal editing.

Previous results for non-streaming transducers: Wagner [3] gives a polynomial time algorithm for computing the optimal edit-distance, namely $u \mapsto \textsc{ed}(u, T)$, given a DFA for T. In [2] it was shown (Section *III.A*) that certain (non-deterministic) distance automata can compute this optimal cost.

Our contributions for non-streaming transducers: We consider the problem of finding a natural class of *deterministic* transducer for computing the optimal edit-distance $u \mapsto \textsc{ed}(u, T)$. We observe that Wagner's algorithm can be reformulated using cost-register automata of [1]. Specifically, one can use the deterministic one-pass transducers with registers that allow parallel updates using the increment and arbitrary-arity minimum operations. We prove that natural restrictions of this model, notably by disallowing use of the minimum operator, do not suffice to compute the optimal edit-distance (Theorem 5). This uses some pumping-like arguments.

Previous results for streaming transducers: In [2] it was also shown that whether there is a streaming transducer (Definition 7) with finite streaming-cost (Definition 8) that repairs strings from R to strings in T is a PTIME-complete problem (the languages R, T are given as DFAs). Moreover, if the cost is finite, then a streaming transducer can be extracted from their proof [2][Theorem 3].

Our contributions for streaming transducers: We consider the problem of repairing strings from source to target language in the case that the streaming-cost is infinite or very large. In this case we fix an edit-distance bound α and ask what is the 'largest fraction' of strings $\eta \in [0, 1]$ that can be repaired within cost α by a streaming transducer. This is called the *partial-repair problem.* We show with an example (Example 12) that although the streaming-cost is infinite, a large fraction (formalised in Definition 11) of the strings from the source language may be edited with small edit distance to belong to the target language. Our main contribution (Theorem 14) is a polynomial time algorithm solving the partial-repair problem, and, if one exists, a construction of a corresponding deterministic streaming transducer. We do this by building and solving a Markov decision process whose value is the largest such η.

2 Non-streaming Transducers

Following Wagner [3], there is a dynamic programming algorithm that given a string $u \in \Sigma^*$ and a DFA for target language $T \subset \Sigma^*$ can compute, in PTIME, the integer $\textsc{cost}(u, T)$ (and a string $t \in T$ with the property that $\textsc{cost}(u, T) = \textsc{ed}(u, t)$). In [2][Section III] it is mentioned that this gives a transducer of fairly low complexity. We formalise this intuition and observe that there is a *deterministic* transducer model that implements Wagner's approach and computes $u \mapsto \textsc{ed}(u, T)$. In fact, the transducers are certain cost register automata (introduced in [1]), which we call *repair-transducers.* A repair-transducer is a DFA with a fixed number of register names K. Write int_k for the value, a natural number, of register $k \in K$. Initially each register contains 0. At each step the DFA reads the next letter from the input string, updates its local state, and makes a parallel update to the registers. The allowed updates may mention the register names, the constant 0, addition by a constant, and the minimum of any number of terms.[1] Once the input string has been read the machine outputs the contents of some of its registers. Thus a repair-transducer realises a function $\Sigma^* \to \mathbb{N}^k$. Formally, an update is a term generated by the grammar:

```
inc-min ::= 0 | int_k  (for k∈K) | inc-min + c  (for c∈ ℕ)
inc-min ::= min(inc-min, inc-min,...,inc-min)
```

If $\nu : K \to \mathbb{N}$ and τ is an inc-min term, write $[[\tau]]_\nu$ for the evaluation of term τ under assignment ν.[2]

Definition 2. *A* repair-transducer $\mathfrak{Tr}$ *is a DFA* (Σ, Q, q_0, δ) *without final states augmented by a finite set* K *of register names, a register update function* $\mu : Q \times \Sigma \times K \to IMT$*, and a final register function* $f : Q \to K$*, where* IMT *is the set of* $inc - min$ *terms. A* configuration *is an element of* $Q \times \mathbb{N}^K$*. The initial configuration is* (q_0, ν_0) *where* $\nu_0(k) = 0$ *for all* $k \in K$*. The* run *on input* $u = u_1 \cdots u_n \in \Sigma^*$ *is the sequence of configurations* $(q_0, \nu_0) \cdots (q_n, \nu_n)$ *such that* $q_{i+1} = \delta(q_i, u_i)$ *(for* $0 \leq i < n$*) and for each* $k \in K$, $\nu_{i+1}(k) := [[\mu(q_i, u_i, k))]]_{\nu_i}$*. The transducer outputs* $\mathfrak{Tr}(u) := \nu_n(f(q_n))$*.*

Proposition 3. *For regular language* T *there is a repair-transducer* $\mathfrak{Tr}$ *computing* $u \mapsto \textsc{cost}(u, T)$*. Moreover, given a DFA for* T *one can build* $\mathfrak{Tr}$ *in PTIME.*

For the proof simply note that the dynamic-programming identities in Wagner's algorithm only use the $+c$ and min operations. Clearly if we disallow use of $+c$ operation then the transducer can't accumulate values and so can't compute $u \mapsto \textsc{ed}(u, T)$. What if we disallow the min operation?

Proposition 4. *There is a language* T *such that every repair-transducer for* T *requires the use of* min *in its update rules.*

Proof. Let $T = 0^* + 1^*$. Suppose there were a repair transducer $\mathfrak{Tr}$ for T with state set Q, register set K, but no use of the min operation. Let $\delta : Q \times \Sigma^* \to Q$ be the transition function (extended to strings) and $M : Q \times \Sigma^* \times \mathbb{N}^K \to \mathbb{N}^K$ the evaluated update function extended to strings. That is: the run of $\mathfrak{Tr}$ starting from (q, ν) on input $u \in \{0, 1\}^*$ ends in $(\delta(q, u), M(q, u, \nu))$. Note that $\mathfrak{Tr}(w)$ is the minimum of the number of 0s in w and the number of 1s in w.

[1] This is similar to the inc-min grammar of [1]. However there they only allow a binary min operation.

[2] Namely, $[[0]]_\nu := 0$, $[[int_k]_\nu := \nu(k)$, $[[\tau + c]] := [[\tau]] + c$, and $[[min(\tau_1, \cdots, \tau_n)]] := \min\{[[\tau_1]], \cdots, [[\tau_n]]\}$.

Fact 1. For every (q,u) there are functions $F_{q,u} : K \to K$ and $\#F_{q,u} : K \to \mathbb{N}$ such that for all ν, k,

$$M(q,u,\nu)(k) = \nu(F_{q,u}(k)) + \#F_{q,u}(k), \tag{1}$$

in words: the kth component of $M(q,u,\nu)$ is equal to the $F_{q,u}(k)$th component of ν plus integer $\#F_{q,u}(k)$. This can be proved by induction on $|u|$ and uses the fact that $\mathfrak{Tr}$ only has $+c$ updates.

Fact 2. For every $F_{q,u} : K \to K$ there exists $N, P \in \mathbb{N}$ such that for all $y \in \mathbb{N}$, $(F_{q,u})^N = (F_{q,u})^{N+Py}$. This follows from the fact that the set of functions $K \to K$ is a finite set closed under composition so apply the pigeonhole principle.

Fact 3. Fix u, q such that $\delta(q,u) = q$, and N, P from Fact 1 applied to $F_{q,u}$. For every $k \in K$ there exists $j \in K$ and $\alpha, \beta \in \mathbb{N}$ such that for all $\nu \in \mathbb{N}^K$ and $y \in \mathbb{N}$, $M(q, u^{N+Py}, \nu)(k) = \beta + y\alpha + \nu(j)$. For the proof use: $\beta := \#F_{q,u^N}(k)$, $j := F^N_{q,u}(k)$, and $\alpha := \#F_{q,u^P}(j)$.

We now apply some pumping arguments. Note that in the lemma if $\alpha = 0$ then pumping doesn't increase the value. In this case call the triple (u,q,k) *flat*. On the other hand if $\alpha > 0$ then pumping increases the value. In this case call the triple (u,q,k) *increasing*. We exploit this dichotomy.

By the pigeonhole principle there exists $q_0 \in Q$ and $c, d \in \mathbb{N}$ such that $\delta(\iota, 0^a) = q_0$, $\delta(q_0, 0^b) = q_0$. Thus $\delta(\iota, 0^{a+bx}) = q_0$ for all $x \in \mathbb{N}$. Similarly, there exists $q_1 \in Q$, $c, d \in \mathbb{N}$ such that $\delta(q_0, 1^c) = q_1$, $\delta(q_1, 1^d) = q_1$. Thus $\delta(q_1, 0^{c+dy}) = q_1$ for all $y \in \mathbb{N}$. Write $w_{x,y} := 0^{a+bx}1^{c+dy}$. Note that $\mathfrak{Tr}(w_{x,y}) = \min\{a+bx, c+dy\}$. Let $k := f(q_1)$ be the register whose value is output when given strings of the form $w_{x,y}$.

Suppose x is much bigger than y (technically: $a + bx > c + d(y+1)$). Then after reading input $w_{x,y}$ register k has value $c + dy$. And after reading input $w_{x,y+1}$ register k has value $c + d(y+1)$. This implies that the triple $(1^d, q_1, k)$ is increasing. On the other hand, suppose x is smaller than y. Then after reading input $w_{x,y}$ register k has value $a + bx$. And after reading input $w_{x,y+1}$ register k has value $a + bx$. This implies that the triple $(1^d, q_1, k)$ is flat. But a triple can not be both increasing and flat. $\square$

We summarise the results of this section:

Theorem 5. *For every regular language $T \subset \Sigma^*$ there is a repair-transducer $\mathfrak{Tr}$ computing* $\textsc{cost} : \Sigma^* \to \mathbb{N}, u \mapsto \textsc{cost}(u,T)$*. Moreover, the models of transducer which disallow the $+c$ operators or the* min *operator cannot, in general, compute this cost.*

3 Streaming Transducers

Here is the generalisation of edit-distance to a source and target language:

Definition 6 (Repair-cost). *[2] Given two languages $R, T \subset \Sigma^*$ define the* repair-cost from R to T *as* $\textsc{cost}(R \rightsquigarrow T) := \sup_{u \in R} \inf_{v \in T} \textsc{ed}(u,v)$.

Note the asymmetry in the definition of repair cost: it expresses the worst-case cost of repairing all strings in R to strings in T. The cost may be finite $\textsc{cost}((ab)^* \rightsquigarrow (ba)^*) = 2$ — just delete the first and last letter of the input — or infinite $\textsc{cost}(a^* \rightsquigarrow (ba)^*) = \infty$ — since $\textsc{ed}(a^{2n}, (ba)^*) = n$.

Definition 7 (Streaming Transducer). *A* streaming transducer *is a device of the form* $\mathfrak{Tr} = (\Sigma, \Sigma_{out}, Q, \delta, q_0, \Omega)$*, where*

- Σ *is a finite* input alphabet*, and* Σ_{out} *is a finite* output alphabet*,*
- Q *is a finite set of* states *and* $q_0 \in Q$ *is an* initial state*,*
- δ *is a* transition function $Q \times \Sigma \to \Sigma_{out}^* \times Q$*, and*
- Ω *is a* final-output function $Q \to \Sigma_{out}^*$.

For every string $u = a_1 \dots a_n$ *in* Σ^**, there is a unique sequence of states* $q_0, q_1, \cdots, q_n$ *and strings* $v_1, \cdots, v_n$ *such that* $\delta(q_i, a_{i+1}) = (v_{i+1}, q_{i+1})$ *for all* $0 \le i < n$. *In this case define the output of* $\mathfrak{Tr}$ *on* u *to be the string* $\mathfrak{Tr}(u) = v_1 v_2 \dots v_n v_{n+1}$ *where* $v_{n+1} = \Omega(q_n)$. *We write* $q_0 \xrightarrow{a_1/v_1} q_1 \xrightarrow{a_2/v_2} \cdots \xrightarrow{a_n/v_n} q_n \xrightarrow{v_{n+1}}$.

Define the output of $\mathfrak{Tr}$ *on language* R *to be the set* $\mathfrak{Tr}(R) = \{\mathfrak{Tr}(u) \mid u \in R\}$.

Definition 8 (Streaming Cost[3]). *[2] For a streaming transducer* $\mathfrak{Tr}$ *and an input string* $u = a_1 \dots a_n \in \Sigma^*$*, if the run of* u *on* $\mathfrak{Tr}$ *is* $q_0 \xrightarrow{a_1/v_1} q_1 \xrightarrow{a_2/v_2} \cdots \xrightarrow{a_n/v_n} q_n \xrightarrow{v_{n+1}}$*, then the* streaming cost *of* $\mathfrak{Tr}$ *on* u *is defined as:*

$$\textsc{cost}_{\mathfrak{Tr}}(u) = |v_{n+1}| + \sum_{i=1}^{n} \textsc{ed}(a_i, v_i)$$

For language R *define* $\textsc{cost}_{\mathfrak{Tr}}(R) := \sup_{u \in R} \textsc{cost}_{\mathfrak{Tr}}(u)$.

If $\mathfrak{Tr}(R) \subset T$ *then say that* $\mathfrak{Tr}$ *is a* streaming transducer from R to T.

Note. $\textsc{ed}(u, \mathfrak{Tr}(u)) \le \textsc{cost}_{\mathfrak{Tr}}(u)$ and so $\textsc{cost}(R \rightsquigarrow \mathfrak{Tr}(R)) \le \textsc{cost}_{\mathfrak{Tr}}(R)$. The streaming-cost is an upper bound on the repair-cost. So if $\textsc{cost}(R \rightsquigarrow T) = \infty$ then there is no streaming-transducer from R to T with finite streaming-cost.

Example 9. [2] Let $\Sigma = \{a, b, c\}$, $R = (a+b)c^*(a^+ + b^+)$ and $T = ac^*a^+ + bc^*b^+$. Then for every $r \in R$ there is $t \in T$ such that $\textsc{ed}(r, t) \le 1$ (correct the first letter if required). However, for every streaming transducer $\mathfrak{Tr}$ from R to T, $\textsc{cost}_{\mathfrak{Tr}}(R) = \infty$. In other words, the cost of repairing all strings in R to strings in T is finite, but is not realisable by a streaming transducer with finite streaming-cost.

In the last example consider a streaming transducer $\mathfrak{Tr}$ that outputs an 'a' and then copies the rest of the input (ie. it sends $(a+b)c^n w$ to $ac^n w$). It correctly repairs strings of the form $(a+b)c^*a^+$ (incurring cost ≤ 1) and incorrectly strings of the form $(a+b)c^*b^+$; that is, it is a streaming transducer from $(a+b)c^*a^+$

[3] In [2] this is called the *aggregate cost* of $\mathfrak{Tr}$ on u.

to T. Then, informally, $\mathfrak{Tr}$ repairs with cost at most 1 half the strings of R to T; and formally $\mathfrak{Tr}$ is a $(\frac{1}{2}, 1)$-streaming transducer (Definition 11). To formalise this we introduce probability measures over infinite strings.

A *distribution on finite set* A is a function $d : A \to [0, 1]$ with $\Sigma_{a \in A} d(a) = 1$. For instance, $d : \sigma \mapsto \frac{1}{|A|}$ is a distribution on A, and $d' : \sigma_1 \cdots \sigma_n \mapsto \prod_i d(\sigma_i)$ is a distribution on A^n. The distributions over A will be denoted $\mathtt{dbn}(A)$. For $u \in A^*$, let $\mathtt{cone}(u) \subset A^\omega$ be the set of infinite strings that have u as a prefix. There is a unique *probability measure* μ_d on the Borel σ-field generated by the cones[4] with the property that the measure of $\mathtt{cone}(u)$ is $d'(u)$.

Example 10. Continuing with Example 9, let d and d' be as above (thus $d'(u) := 3^{-|u|}$). The probability (wrt. μ_d) that every prefix of an infinite string is in $(a + b)c^* a^+$ conditioned on the infinite string having a prefix in $R = (a+b)c^*(a^+ + b^+)$ is $\frac{1}{2}$.

Definition 11 ((η, α)-streaming transducer). *Fix regular languages R, T, and a streaming transducer $\mathfrak{Tr}$ from R to T, and a non-negative integer α. Say that infinite string $u = a_0 a_1 a_2 \ldots$*

1. is in need of R-repair *if there exists n so that $a_0 \ldots a_n$ is in R;*
2. *is* $\langle R, T\rangle$-repairable by $\mathfrak{Tr}$ within α *if for all n with $a_0 \ldots a_n \in R$ the cost $\text{COST}_{\mathfrak{Tr}}(a_0 \ldots a_n)$ is at most α and $\mathfrak{Tr}(a_0 \ldots a_n) \in T$.*

Say that $\mathfrak{Tr}$ is an (η, α)-streaming transducer (from R to T) *if η is the probability that u is $\langle R, T\rangle$-repairable by $\mathfrak{Tr}$ within α conditioned on u being in need of R-repair. Here probabilities are taken with respect to μ_d induced by the measure on cones d' itself determined by $d : w \mapsto |\Sigma|^{-|w|}$ where Σ is the alphabet of R. When R and T are fixed we may not mention them.*

We give an example where η is close to 1:

Example 12. Let $\Sigma = \{a, b, c, d\}$ and $R_k := (\{a, b\}^k \setminus b^k)c^+ \cup b^k d^+$ and $T = (a + b)^* c^+$. Fix k and note that $\text{COST}(R_k \rightsquigarrow T) = \infty$ since $\text{ED}(b^k d^n, T) = n$ (as T does not accept any string with d in it). However, there exists an (η, α)-streaming transducer with $\alpha = 0$ and $\eta = 1 - \frac{1}{2^k}$ which operates as follows: it copies the first k letters and then outputs a 'c' for every remaining input letter. The only strings in R_k which it cannot repair within cost 0 are the ones of the form $b^k d^+$.

Partial-repair Problem. The *bounded-repair problem* is, given DFAs for R and T to decide whether or not there exists a streaming transducer $\mathfrak{Tr}$ from R to T such that $\text{COST}_{\mathfrak{Tr}}(R)$ is finite. It is proved in [2] that the bounded repair problem is PTIME-complete. Moreover, if it exists, a streaming transducer can be constructed quite easily from their proof.

[4] The Borel σ-field is defined as the least collection of subsets of A^ω containing the cones and closed under countable union and complementation. Sets in the σ-field are called measurable. All our sets in this paper are measurable.

Question 13. Suppose $\textsc{cost}(R \rightsquigarrow T) = \infty$, or $\textsc{cost}_{\mathfrak{Tr}}(R)$ is ∞ or just very large for every streaming-transducer $\mathfrak{Tr}$ from R to T. How to transform R to T?

Our proposal is, given R, T and allowed cost α, to construct a streaming transducer $\mathfrak{Tr}$ that, roughly, repairs as many strings as possible. Formally this means solving the *partial-repair problem*: compute the largest η for which there exists a (η, α)-streaming transducer from R to T; and compute the corresponding transducer. The main theorem of this section states that we can do this in PTIME:

Theorem 14 (partial-repair problem). *Given DFAs for R and T, and positive integer α, given in unary, one can compute, in PTIME, the largest $\eta \in [0, 1]$ for which there exists an (η, α)-streaming transducer sending R to T.*[5] *Moreover, we can build an (η, α)-streaming transducer from R to T in PTIME.*

The rest of the paper is devoted to a proof of this theorem.

3.1 Tools for Theorem 14

Definition 15 (MC). *A* Markov chain *M is a tuple (Q, Δ, ι) where*

- *Q is a finite set of states,*
- *$\Delta : Q \to \mathtt{dbn}(Q)$ gives the* transition probabilities*, and*
- *$\iota \in \mathtt{dbn}(Q)$ is the* initial distribution*.*

The edges *E consist of pairs (q, q') such that $\Delta(q)(q') > 0$. A* path *$q_1 q_2 \cdots$ of M is a (finite or infinite) sequence of states such that $\iota(q_1) > 0$ and successive states q_i, q_{i+1} satisfy E.*

Write Ω_M for the set of infinite paths in M. Form a topology on Ω_M by taking as basis the sets of the form $\mathtt{cone}(x)$ where $\mathtt{cone}(x)$ consists of all infinite paths in M that start with the finite path x. Define the *probability in M of a path* $q_1 \cdots q_n \in Q^+$ as $\iota(q_1) \times \prod_{1 \leq i < n} \Delta(q_i)(q_{i+1})$. Define μ_M on $\mathtt{cone}(x)$ as the probability in M of path x. Then μ_M can be uniquely extended to the Borel σ-field generated by the open sets. Write $\mathtt{Pr}_M$ for the unique probability measure (over Ω_M) extending μ_M.

Definition 16 (Labelled MC). *A Markov chain $M = (Q, \Delta, \iota)$ is Σ-*labelled *if for each $q \in Q$ the edges out of q (ie. $E(q) := \{(q, q') : E(q, q')\}$) are in bijection with Σ.*

Being labelled means that every state q has exactly $|\Sigma|$ edges, and each edge goes to a different state.

Example 17 (Uniform MC U_Σ). Let $U = U_\Sigma$ have states Σ, transition from σ to σ' labelled σ' with probability $\frac{1}{|\Sigma|}$, initial distribution sends σ to $\frac{1}{|\Sigma|}$. Then U is a Σ-labelled MC. The probability of $u \in \Sigma^+$ is equal to $|\Sigma|^{-|u|}$. Thus the measure $\mathtt{Pr}_U$ agrees with μ_d on the cones $\mathtt{cone}(u)$. Hence $\mathtt{Pr}_U$ and μ_d agree on the measurable subsets of Σ^ω.

[5] That is, η is a rational and the algorithm computes a representation for it in PTIME.

The following lemma is standard:

Lemma 18. *There is a PTIME algorithm that given a MC M and a set of states A computes the probability that a path in M reaches A.*

The following definition annotates a MC by the states of a DFA.

Definition 19 ($M \oslash_{mc} D$). *For Σ-labelled Markov chain M and DFA D over alphabet $\Sigma \times Q_M$ define the Σ-labelled Markov chain $M \oslash_{mc} D$ as follows:*

- *The state set is $Q_M \times Q_D$.*
- *Suppose there is an edge $E_M(m, m')$ labelled σ. Then there is an edge from (m, d) to $(m', \delta_D(d, (\sigma, m')))$ with probability $\Delta(m)(m')$ and label σ.*
- *the initial distribution sends (m, d) to $\iota_M(m)$ if d is the initial state of D, and to zero otherwise;*

As a degenerate case, in case D has alphabet Σ then write $M \oslash_{mc} D$ to mean $M \oslash_{mc} F$ where F has the same state set as D, the same initial state, has alphabet $\Sigma \times Q_M$, and sends, for all m, state q on input (σ, m) to $\delta_D(q, \sigma)$.

Note. It can be checked that the object defined is indeed a Σ-labelled Markov chain. We point out that in an edge (m, d) to (m', d') labelled σ, the state d' depends directly on m' — not m — and σ.

Let M and D be as in the definition. Every path $m_1 m_2 \cdots$ of M induces a unique sequence of labels $\sigma_1 \sigma_2 \cdots$ such that the edge from (m_i, m_{i+1}) is labelled σ_i which itself induces a unique sequence $d_1 d_2 \cdots$ of states of D satisfying $d_{i+1} = \delta_D(d_i, (\sigma_i, m_{i+1}))$ where d_1 is the initial state of D. Let

$$\rho : m_1 m_2 \cdots \mapsto (m_1, d_1)(m_2, d_2) \cdots$$

be the *annotation map.* Note that since D is a DFA ρ is a bijection between paths in M and paths in $M \oslash_{mc} D$.

Lemma 20. *Let M be a Σ-labelled MC, D a DFA over $\Sigma \times Q_M$. Then for every measurable $X \subset (Q_M)^\omega$, $\mathtt{Pr}_M(X) = \mathtt{Pr}_{M \oslash_{mc} D}(\rho(X))$.*

For the proof it is enough to consider X of the form $\mathtt{cone}(x)$.

Example 21. Let R be a DFA over Σ with final states F_R and $U = U_\Sigma$ the uniform Markov chain. The lemma says that $\mathtt{Pr}_U$ of the set of paths in need of R-repair equals the probability in $\mathtt{Pr}_{U \oslash_{mc} R}$ of the set of paths that reach a state of the form $\Sigma \times F_R$.

Definition 22. *A* Markov decision process *is a tuple $((V, E), (V_{dec}, V_{rand}), \mu_\iota, \mu)$ where*

- *(V, E) is a directed graph, V_{dec}, V_{rand} partition V,*
- *$\mu \colon V_{rand} \to \mathtt{dbn}(V_{dec})$ is the edge distribution,*
- *$\mu_\iota \in \mathtt{dbn}(V_{rand})$ is the initial distribution,*

– *for* $u \in V_{rand}, (u, v) \in E$ *iff* $\mu(u)(v) > 0$,
– *for* $v \in V$, $E(v)$ *(the out-going edges from* v*) is non-empty.*

We say that the vertices V_{dec} *belong to the* ***decider****, while the vertices* V_{rand} *belong to the* ***randomizer****. A* play *is a path in* (V, E)*.*

We think of a labelled MDP just as a labelled MC with the addition that decider's edges are labelled by elements from a set $\mathtt{Act}$. Formally:

Definition 23 (Labelled MDP). *An MDP is* $(\Sigma, \mathtt{Act})$-labelled *if*

– *for each* $v \in V_{rand}$ *the edges from* v *are in bijection with* Σ*, and*
– *for each* $v \in V_{dec}$ *each edge from* v *is labelled by an element of* $\mathtt{Act}$*.*

Note (identification). Suppose $|E(u)| = 1$ for all $u \in V_{dec}$. Then a $(\Sigma, \mathtt{Act})$-labelled MDP can be naturally viewed as Σ-labelled MC as well as a streaming transducer with input alphabet Σ and output values taken from $\mathtt{Act}$.[6] We call this *identification* and write things like "this MDP is the same, modulo the identification, as that MC".

Definition 24 (Strategy in an MDP). *A* strategy σ *for the decider is a function* $\sigma \colon V^* \cdot V_{dec} \to V$ *such that for all* $w \in V^*$ *and all* $v \in V_{dec}$ *we have* $\sigma(w \cdot v) \in E(v)$*. A* memoryless strategy *for the decider is independent of the history and depends only on the current state, and can be described as a function* $\sigma : V_{dec} \to V$*. A* finite-state strategy *for the decider is one induced by a DFA* (Q, δ, ι) *over input alphabet* V *and output function* $\theta : V \times Q \to V$ *as follows:* $\sigma(wv) := \theta(v, \delta(\iota, w))$*.*

As usual, applying a strategy s to an MDP G results in a MC, which we write $G[\mathsf{s}]$.

Definition 25. *Let* s *be a finite-state strategy in* G*. Write* $\mathfrak{Tr}_{\mathsf{s}}$ *for the streaming transducer associated with* $G[\mathsf{s}]$*. Note that* $\mathfrak{Tr}_{\mathsf{s}}$ *has input alphabet* Σ *and outputs elements from* $\mathtt{Act}$*.*

We now define a certain interleaving of a MC and an NFA yielding an MDP. The idea is that the MC determines the allowed moves of the randomizer while the NFA determines the allowed moves of the decider.

Definition 26 ($M \oslash_{mdp} N$). *Suppose* M *is a* Σ*-labelled Markov chain and* N *is an NFA over alphabet* $\Sigma \times \mathtt{Act}$*. Define a* $(\Sigma, \mathtt{Act})$*-labelled MDP* $M \oslash_{mdp} N$ *as follows:*

– *the randomizer's nodes are* $Q_M \times Q_N$*,*
– *the decider's nodes are* $Q_M \times \Sigma \times Q_N$*;*
– *if in* M *there is an edge from* m *to* m' *with label* σ *and probability* x*, then for all* n *there is an edge in* $M \oslash_{mdp} N$ *from* (m, n) *to* (m', σ, n) *with label* σ *and probability* x*;*

[6] Later $\mathtt{Act}$ will be a set of strings output by a streaming transducer.

– *if in N there is a transition from n to n' labelled $\sigma \in \Sigma$ and $a \in$ `Act`, then for all m there is an edge from (m, σ, n) to (m, n') labelled a;*
– *the initial distribution sends (m, n) to $\iota_M(m)$ if n is the initial state of N, and to 0 otherwise.*

Note. In case D is a DFA then $M \oslash_{mdp} D$ (with the `Act`-labelling removed) is, modulo identification, the Σ-labelled MC $M \oslash_{mc} D$.

Lemma 27. *If* $\mathtt{s}$ *is a finite-state strategy then the MC $(M \oslash_{mdp} N)[\mathtt{s}]$ is, modulo identification, the same as $M \oslash_{mc} D$ for some DFA D.*

An *objective Φ* for a game graph is a subset of plays. We consider two types of objectives, reachability and safety objectives. Given a set $X \subset V$ of nodes, the reachability objective $\mathtt{reach}(X)$ requires that some vertex in X be visited, and dually, the safety objective $\mathtt{safe}(X)$ requires that only vertices in X be visited. *Solving an MDP for objective Φ* means finding a strategy $\mathtt{s}$ such that, amongst all possible strategies, the probability in the chain $G[\mathtt{s}]$ of Φ is maximised. We call this maximal probability the *value* of the MDP for objective Φ. The following lemma is a slight variation on the standard problem of solving MDPs with reachability objectives.

Lemma 28. *There is a PTIME algorithm that computes the value (and strategy) of an MDP whose objective is a boolean combination of reachability objectives.*

3.2 Proof of Theorem 14

From DFAs R and T and non-negative integer α we construct an MDP $G_{R,T,\alpha}$ and objectives $\mathtt{reach}(\mathtt{Ob}_R)$ and $\mathtt{safe}(\mathtt{Ob}_S)$ such that the following two quantities are equal: 1) the maximum probability over all strategies of $\mathtt{safe}(\mathtt{Ob}_S)$ conditioned on $\mathtt{reach}(\mathtt{Ob}_R)$; 2) the largest $\eta \in [0, 1]$ for which there exists an (η, α)-streaming transducer sending R to T. We now provide the construction (in I), then show how to compute the value (in II), and finally prove that the value is equal to the required conditional probability (in III).

I. Construction of MDP $G_{R,T,\alpha}$. The MDP is constructed in three steps:

Step 1. From the DFA $R = (Q_R, \Sigma, \delta_R, q_{0R}, F_R)$ construct the Σ-labelled Markov chain $UR := U_\Sigma \oslash_{mc} R$, as in Example 21. Its state set is $\Sigma \times Q_R$.

Step 2. From the DFA $T = (Q_T, \Sigma, \delta_T, q_{0T}, F_T)$ and non-negative integer α construct an NFA (without final states) T_α over alphabet $\Sigma \times \Sigma^*$ that simulates the possible repairs of the input string. It does this by storing the allowed number of edit operations left.The non-determinism will corresponds to Decider's choices in the MDP. First we need some notation. Write BEST-STR(q, q', σ) for some fixed string w (say the length-lexicographically least) amongst those for which ED(w, σ) is minimal with the property that $\delta_T(q, w) = q'$. Now, define T_α as follows:

– the states are $Q_T \times \{\perp, 0, 1, \cdots, \alpha\}$ (here $\perp$ means we have failed to repair the input string);

- the initial state is (q_{0T}, α) (meaning initially there are α edit operations available);
- on input (σ, σ) there is a transition from $(q, \perp)$ to $(\delta_T(q, \sigma), \perp)$, (ie. once we have failed to repair, just copy the input to the output);
- on input $(\sigma, \textsc{best-str}(q, q', \sigma))$ there is a transition from (q, n) to (q', m) where
$$m = n - \textsc{ed}(\textsc{best-str}(q, q', \sigma), \sigma)$$
if this quantity is non-negative, and otherwise $m = \perp$.

Step 3. Define the Σ-labelled MDP $G_{R,T,\alpha}$ as $\oslash_{mdp}$-product of the Markov chain UR and NFA T_α.

Notation. Every node in V_{rand} is of the form $(q, t, n) \in Q_{UR} \times Q_T \times \{\perp, 0, 1, \cdots, \alpha\}$. Every node in V_{dec} is of the form $(q, \sigma', t, n) \in Q_{UR} \times \Sigma \times Q_T \times \{\perp, 0, 1, \cdots, \alpha\}$. The second component of the element $q \in Q_{UR} := \Sigma \times Q_R$ is called the Q_R-*component* of (q, t, n) and of (q, σ', t, n).

Objectives. We introduce two objectives. Let $\mathtt{Ob}_R$ be the set of states of $G_{R,T,\alpha}$ whose Q_R-component is in F_R. Let $\mathtt{Ob}_S$ be the set of states of $G_{R,T,\alpha}$ such that if the Q_R-component is in F_R then both $t \in F_T$ and $n \neq \perp$. The two objectives are $\mathtt{reach}(\mathtt{Ob}_R)$ and $\mathtt{safe}(\mathtt{Ob}_S)$.

II. Computing the Value of $\boldsymbol{G_{R,T,\alpha}}$. Given R,T and α, the value η_* of $G_{R,T,\alpha}$ is defined as the maximum, over all strategies $\mathtt{s}$, of the conditional probability,

$$\mathtt{Pr}_{G_{R,T,\alpha}[\mathtt{s}]}\left(\mathtt{safe}(\mathtt{Ob}_S) \mid \mathtt{reach}(\mathtt{Ob}_R)\right) = \frac{\mathtt{Pr}_{G_{R,T,\alpha}[\mathtt{s}]}\left(\mathtt{safe}(\mathtt{Ob}_S) \cap \mathtt{reach}(\mathtt{Ob}_R)\right)}{\mathtt{Pr}_{G_{R,T,\alpha}[\mathtt{s}]}\left(\mathtt{reach}(\mathtt{Ob}_R)\right)}$$

Proposition 29. *The value of the mdp $G_{R,T,\alpha}$ is computable in PTIME and can be realised by a memoryless strategy.*

For the proof observe that the value of $\mathtt{reach}(\mathtt{Ob}_R)$ is independent of the strategy $\mathtt{s}$ chosen by the decider. This is so because $\mathtt{s}$ does not have any effect on the Q_R-component of the state of $G_{R,T,\alpha}$. Thus the value of $\mathtt{reach}(\mathtt{Ob}_R)$ can be easily calculated (fix any memoryless strategy and apply Lemma 18). So, we just need to find the value of the objective $\mathtt{safe}(\mathtt{Ob}_S) \cap \mathtt{reach}(\mathtt{Ob}_R)$. By Lemma 28 this can be computed, and the required strategy is memoryless.

III. Existence of an Optimal Streaming Transducer. Fix R, T and α, and let η_* be the value of the MDP $G_{R,T,\alpha}$ for the property $\mathtt{safe}(\mathtt{Ob}_S)$ conditioned on $\mathtt{reach}(\mathtt{Ob}_R)$. In this section Proposition 31 immediately implies what we want, namely: 1) there is an (η_*, α)-streaming transducer from R to T; 2) there is no (η, α)-streaming transducer from R to T with $\eta > \eta_*$.

Lemma 30. *Let $\mathtt{s}$ be a finite-state strategy for $G_{R,T,\alpha}$ and $\mathfrak{Tr}_{\mathtt{s}}$ the corresponding streaming transducer. Then the probability in MC $G_{R,T,\alpha}[\mathtt{s}]$ of $\mathtt{safe}(\mathtt{Ob}_S) \cap \mathtt{reach}(\mathtt{Ob}_R)$ divided by the probability of $\mathtt{reach}(\mathtt{Ob}_R)$ is equal to the probability that a string in need of R-repair is $\langle R, T\rangle$-repaired by $\mathfrak{Tr}_{\mathtt{s}}$ within α.*

Proof. By Lemma 27 $G_{R,T,\alpha}[\mathsf{s}]$ is a MC of the form $UR \oslash_{mc} D$ for some DFA D. Thus the probability of $\mathtt{reach}(\mathtt{Ob}_R)$ in $G_{R,T,\alpha}[\mathsf{s}]$ is equal to the probability in $UR \oslash_{mc} D$ of reaching a state whose Q_R-component is in F_R. Note that the annotation map ρ preserves the property that a path reaches a state whose Q_R-component is in F_R. By Lemma 20 the latter is equal to the probability in UR that a path reaches a state whose Q_R-component is in F_R. By Example 21 this is equal to the probability that that an infinite string is in need of repair. Similarly it can be shown that the probability of $\mathtt{safe}(\mathtt{Ob}_S)$ in $G_{R,T,\alpha}[\mathsf{s}]$ is equal to the probability in that an infinite string is $\langle R, T\rangle$-repairable by $\mathfrak{Tr}_{\mathsf{s}}$ within α; and the same for the intersection. □

Proposition 31. *1. From a memoryless strategy* s *in* $G_{R,T,\alpha}$ *with probability (of the conditional objective)* η *one can construct an* (η, α)*-streaming transducer from* R *to* T*.*

2. From an (η, α)*-streaming transducer* $\mathfrak{Tr}$ *from* R *to* T *one can construct a strategy* $\mathsf{s}_{\mathfrak{Tr}}$ *in* $G_{R,T,\alpha}$ *with value* $\geq \eta$*.*

Proof. The first item is immediate from Lemma 30.

For the second, a transducer $\mathfrak{Tr}$ gives rise to the following strategy $\mathsf{s}_{\mathfrak{Tr}}$: suppose $\rho \in V^*$ is a play ending in $(q, \sigma, t, n) \in V_{dec}$ where q is a state of UR and t of T_α. Let $\mathtt{in}(\rho)$ be the input (ie. the letters that Randomizer has chosen) and note that it ends in σ. The strategy $\mathsf{s}_{\mathfrak{Tr}}$ sends ρ to node $(q, t', n') \in V_{rand}$, where $t' = \delta_T(q_{0T}, \mathfrak{Tr}(\mathtt{in}(\rho)))$ where δ_T is the transition function and q_{0T} the initial state of the DFA for T. Note that $\mathsf{s}_{\mathfrak{Tr}}$ is a finite-state strategy. So let $\mathfrak{Tr}'$ be the transducer associated with strategy $\mathsf{s}_{\mathfrak{Tr}}$ and apply Lemma 30. Then the probability in $G_{R,T,\alpha}[\mathsf{s}_{\mathfrak{Tr}}]$ of $\mathtt{safe}(\mathtt{Ob}_S) \cap \mathtt{reach}(\mathtt{Ob}_R)$ divided by the probability of $\mathtt{reach}(\mathtt{Ob}_R)$ is equal to the probability that a string in need of R-repair is $\langle R, T\rangle$-repaired by $\mathfrak{Tr}'$ within α. It is required to show that this latter probability is at least η. For this it is sufficient to show that if a string is repaired by $\mathfrak{Tr}$ then it is repaired by $\mathfrak{Tr}'$. But this is the case because although both $\mathfrak{Tr}$ and $\mathfrak{Tr}'$ suggest the same next state for a given input string, say from (q, σ, t, n) to (q, t', n'), they possibly differ on the output string, say u and v. In particular, $\textsc{ed}(\sigma, u) \geq \textsc{ed}(\sigma, v) := \textsc{best-str}(t, t', \sigma)$ by the definition of $G_{R,T,\alpha}$. □

References

1. Alur, R., D'Antoni, L., Deshmukh, J.V., Raghothaman, M., Yuan, Y.: Regular functions, cost register automata, and generalized min-cost problems. CoRR abs/1111.0670 (2011)
2. Benedikt, M., Puppis, G., Riveros, C.: Regular repair of specifications. In: LICS, pp. 335–344 (2011)
3. Wagner, R.A.: Order-n correction for regular languages. Commun. ACM 17(5), 265–268 (1974)

Execution Information Rate for Some Classes of Automata

Cewei Cui[1], Zhe Dang[1], Thomas R. Fischer[1], and Oscar H. Ibarra[2]

[1] School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164, USA
{ccui,zdang,fischer}@eecs.wsu.edu
[2] Department of Computer Science
University of California, Santa Barbara, CA 93106, USA
ibarra@cs.ucsb.edu

Abstract. We study the Shannon information rate of accepting runs of various forms of automata. The rate is therefore a complexity indicator for executions of the automata. Accepting runs of finite automata and reversal-bounded nondeterministic counter machines, as well as their restrictions and variations, are investigated and are shown, in many cases, with computable execution rates. We also conduct experiments on C programs showing that estimating information rates for their executions is feasible in many cases.

Key words: information rate, automaton, execution.

1 Introduction

A program, when it runs, will demonstrate an execution path. From a programmer's view, the complexity of the paths is a good indicator of the program's semantic complexity, when one treats the program as a white-box. Such an indicator is useful in software testing, where a set of test cases chosen according to a coverage criterion can again be re-evaluated to see how much of the semantics are "covered".

Automata have been a fundamental and universal model for all modern programs. Therefore, providing a complexity measure for executions of automata may provide insights to real-world programs. In theory, an execution or a run, being a sequence of instructions, is a word. In this way, one may collect all the accepting runs (on all the possible input words) of an automaton into a language L. Then, what would be a good complexity measure for L? Traditional automata theory measures a language's complexity from the computing resources needed by a device to recognize a word in L; i.e., time/space, storage (finite memory, pushdown stack, counters etc). However, such measurements are not obviously relevant to the aforementioned semantic complexity of programs. In fact, there has already been a fundamental notion called information rate, due to Shannon [17], to characterize such complexity, which has been confirmed at least in coding theory [6]. It measures, asymptotically, the bit rate needed to losslessly code words of length n in a language L. In this paper, we study various classes of automata whose execution information rates are computable.

We first study finite automata (NFA). It turns out that, as far as a two-way NFA (2NFA) is concerned, the problem of computing its execution information rate is in general open, even when the 2NFA does not have ϵ-moves, sweeps from left to right and back, and operates only on unary input. The root of the difficulty inherently comes from the fact that accepting runs of such two-way automata are not necessarily regular nor context-free. Results [15,19,2] on computing information rates for formal languages are very limited, not to say for non-context-free languages. We show that, when the 2NFA is finite-crossing, the rate is computable. We also show that, through a rather complex proof, when the aforementioned restricted 2NFA further required to be bounded-crossing (roughly speaking, on an input, the 2NFA can keep sweeping for an unbounded number of times), the rate is finally computable. Though the result is very restricted, it reveals, on the other hand, the difficulty of establishing computability of information rate for non-context-free languages.

As for automata with infinite states, we study reversal-bounded nondeterministic counter machines (NCM), and show that these machines, as well as many variations, have computable execution information rates. The proofs rely on our recent fundamental result [7] concerning the computable information rate for the language accepted by a deterministic reversal-bounded counter machine. Whether the fundamental result still holds for nondeterministic machines is open.

Automata are programs. Providing a complexity metric for programs has been a research issue with a long history; i.e., [11,10,3,4,18]. But most existing research measures the complexity on its code, with only indirect relationships to its execution complexity. Clearly, our execution information rates are not limited to automata; they could be generalized to programs as well. Indeed, we also conduct experiments on C programs showing that estimating information rates for their executions is feasible in many cases. However, because in general programs are Turing-complete, it is a new research issue on how to estimate the execution information rates for large programs in practice.

2 Execution Information Rate for Finite Automata

An automaton is a device that works on a given input word (when it is 1-tape automaton). Its program is a set of distinct instructions. It may be equipped with additional storage; e.g., a stack, one or more counters. A run of the automaton is simply a sequence of instructions. When coded properly, an instruction can be understood as a symbol; in this case, the run is a word. A run is a execution behavior of the automaton in the sense that if we treat an automaton as a C program, then the run is simply a representation of a sequence of instructions in the C program may or may not be executed successfully.

A run may not be valid; i.e., does not correspond to an actual execution of the automaton. For instance, a run that ends up with state s, followed by one more instruction saying that to make a move from state $s' \neq s$, is certainly invalid. Hence, a valid run is one that will not make the automaton crash. A run is accepting if it is an actual run, starting from the initial state, of the automaton on some input word such that the last instruction in the run ends up with an accepting state (encoded in the instruction).

Let M be an automaton and $L(M)$ be the language accepted by M. We use $R(M)$ to denote the set of all accepting runs of M; i.e., $R(M) = \{\alpha$: for some $w \in L(M)$, α is an accepting run when M works on $w.\}$ In particular, when M is nondeterministic, M may have multiple accepting runs on w.

Clearly, when M is an NFA, $R(M)$ is regular. When M is a 2NFA, it is an NFA but can make turns on the input. In this case, the set $R(M)$ can be defined accordingly. However, when M runs, after making a turn, the input symbols read must be the same input symbols read before the turn. Observe that, even though a 2NFA M can be simulated by an NFA, $R(M)$ is not necessarily regular nor context-free.

A 2NFA M is finite-crossing if, when it runs, the input head crosses any cell on the input tape up to a given constant number of times. A 2NFA M is k-turn if it makes at most k turns in any execution on any input word. M is finite-turn if it is k-turn for some k. A finite-turn 2NFA is also finite-crossing. In either case, even though M can be simulated by an NFA, the set $R(M)$, again, is not necessarily regular nor context-free.

The aforementioned variations of NFAs can be found in a standard textbook [12]. When the automata are deterministic, we use 'D' in place of 'N'; e.g., DFA.

Let Σ be a nonempty and finite alphabet and $L \subseteq \Sigma^*$ be a language. For a number n, we use $S_n(L)$ to denote the number of words in L whose length is n. The *information rate* λ_L of L is defined as

$$\lambda_L = \lim_{n \to \infty} \frac{\log S_n(L)}{n}.$$

Where the limit does not exist, we take the upper limit, which always exists for a finite alphabet. By convention, $\log 0 = 0$. Throughout this paper, the logarithm is to base 2. This definition was proposed by Shannon [17], to characterize the number of bits per symbol needed to losslessly encode a word in L, and by Chomsky and Miller [5] to measure a complexity of a regular language. In a recent paper [7], we show that, for some classes of formal languages, the information rate is computable, using a purely automata-theoretic approaches. When an automaton is thought of a black-box, an accepted word is an externally observable "behavior" of the automaton. In this sense, the information rate of the language accepted by the automaton is to measure the complexity of the automaton's externally observable behaviors. In this paper, we study the information rate of $R(M)$ for an automaton M. The rate is also called the execution information rate of M. The rate is important in practice as well since, when a C program is modeled as an automaton, it provides a way to estimate the number of test cases (e.g., execution paths) needed for execution testing the C program.

We need the following fundamental result.

Theorem 1. *The information rate of a regular language is computable [5].*

When M is an NFA, clearly, $\lambda_{R(M)}$ is computable since $R(M)$ is a regular language. Notice that, in our setting, an instruction is succinctly encoded as a symbol (with length 1), but the full instruction can always be recovered by a finite table look-up. In particular, when M is a DFA, $\lambda_{L(M)} = \lambda_{R(M)}$.

Unfortunately, when M is a 2NFA, currently we do not know whether $\lambda_{R(M)}$ is computable. We are going to study some restricted classes of 2NFAs.

Theorem 2. *The execution information rate of a finite-crossing (and hence finite-turn) 2NFA is computable.*

The proof idea of Theorem 2 is to use the standard crossing sequence technique and introduce a length-preserving and one-to-one encoding of an execution.

As we have mentioned earlier, we do not know whether the execution information rate of a 2NFA M is computable. However, one can always restrict the M to be finite-crossing and hence the rate computed (using Theorem 2) for the restricted M can be regarded as a lower approximation of the original $\lambda_{R(M)}$.

There is a special case for a non-finite-crossing 2NFAs where the rate is computable. We will study it in below.

A 2NFA is sweeping if it does not have ϵ-instructions (i.e., will read a symbol on every move), and, furthermore, every right-to-left turn happens when the head is at the left end of the tape and every left-to-right turn happens when the head is at the right end of the tape. We shall emphasize again that the accepting runs of a sweeping 2NFA are not necessarily regular nor context-free. Also, even every 2NFA can be made one-way equivalently in terms of language acceptance, doing this may not maintain the rate of runs of the original 2NFA.

One can imagine a run of a sweeping 2NFA is a multitrack word (or, more precisely, a two-dimension word): An accepting run always forms a rectangle with the width n being the length of the input word and the height k being the number of sweeps that are made in the run; we shall call such a run to be a (n, k)-run. Clearly, the total length of the run is $T = 2nk$. (Without loss of generality, we assume that the head of M returns to the left end of the tape when it accepts. Each sweep is a round-trip: from the left end of the tape to the right end, and back to the left end.)

We say that a sweeping 2NFA M with m states is g-crossing-bounded if g is a monotonic function with $g(0) \geq m$ and $\lim g(n) \to \infty$ such that every accepting (n, k)-run of M is g-crossing-bounded; i.e., satisfying $k \leq g(n)$. Intuitively, it says that such an M will not execute a run that keeps sweeping the input for an unbounded number of times. However, M is not necessarily finite-crossing since the bound is not uniform in the length n of the input. For instance, when $g(n) = n+m$, an accepting run of M always sweeps for at most $n+m$ times when n is large. This restriction is outside the automaton, of course, since the 2NFA does not have the memory to count the unbounded number of crossings. That is, there is a stand alone monitor to supervise a run; when the number of crossings exceeds the bound, the automaton will be shut down without accepting. We use $R_g(M)$ to denote the set of all accepting runs of M (that are g-crossing-bounded). M is crossing-bounded if it is g-crossing-bounded for some g.

A unary 2NFA is one works on unary input with left and right endmarkers. Now, we have the following result.

Theorem 3. *The execution information rate of a crossing-bounded sweeping unary 2NFA is computable.*

Proof. Let M be a g-crossing-bounded sweeping unary 2NFA. Before we start further with the proof, some definitions are needed. An $s's$-sweep is a run that starts from state s' with the head at the left end of the tape and makes exactly one turn (at the right end of the tape) before entering the left end of the tape again at state s. In below, when we say that two runs are concatenated, we implicitly assume that the ending state of the first run is the same of the starting state of the second run. An s-loop is a run starting with state s and ending with state s, and is a concatenation of one or more sweeps. A simple-s-loop is a s-loop where there is no s'-loop, for any s', inside. A simple-s-loop can not contain more than m (the number of states in M) sweeps, and hence is finite-crossing.

Let α be an accepting run on unary input word w (the w is encoded in the run). Clearly, the α can be expressed as a concatenation of a number of sweeps:

$$\beta_{s^0 s^1} \cdots \beta_{s^{k-1} s^k}, \tag{1}$$

for some distinct states $s^0, \cdots, s^q$, satisfying: $\beta_{s^0 s^1}, \cdots, \beta_{s^{k-1} s^k}$ are $s^0 s^1$-sweep, $\cdots$, $s^{k-1} s^k$-sweep, respectively, and, s^0 is the initial state, s^k is the accepting state of M (without loss of generality, we assume that the initial and the accepting states are distinct in M). In particular, $s^0 \cdots s^k$ is called the state sequence of α. Assume that α contains more than m sweeps (i.e., $k > m$), and hence it contains one or more loops. One by one, we can delete simple loops from α, until only one simple loop, say, a simple-s-loop, left. Suppose that the state sequence (with length not longer than $m + 1$) of the simple-s-loop is τ, and after we further delete the simple-s-loop, we obtain an accepting run, with its state sequence (with length not longer than m) denoted by η, that does not contain any loops. In this case we say that the original α has signature $\theta = (\eta, \tau)$. Note that the sequence η contains exactly one appearance of state s and the sequence τ must start and end with the state s. To emphasize this fact, we use (θ, s) to denote the signature instead. There are only finitely many choices of signatures, and one α may corresponds to multiple signatures. We use $L^{\text{simple}}_{(\theta,s)}$ to denote the set of all simple-s-loops, such that, for each such simple-s-loop, there is an accepting run α containing exactly one loop, which is the simple-s-loop, and having signature (θ, s). Every run in $L^{\text{simple}}_{(\theta,s)}$ is finite-crossing, and hence, from Theorem 2, the information-rate $\lambda_{L^{\text{simple}}_{(\theta,s)}}$ is therefore computable. (A run in $L^{\text{simple}}_{(\theta,s)}$; it is a simple-$s$-loop contained in an accepting run and, after deleting all the loops but the simple-s-loop, the accepting run is finite-crossing. We can prove that the simple-s-loop is indeed part of a finite crossing and accepting run by running another constructed NFA. We omit the details.)

We use λ^* to denote the maximum of all $\lambda_{L^{\text{simple}}_{(\theta,s)}}$; i.e.,

$$\lambda^* = \max_{(\theta,s)} \lambda_{L^{\text{simple}}_{(\theta,s)}} \tag{2}$$

and use (θ^*, s^*) for the signature that reaches the maximum in (2). Notice that on an input size of n, a run in $L^{\text{simple}}_{(\theta,s)}$ has fixed length $2|(\theta,s)|n$, where $|(\theta,s)|$, the length of τ specified inside θ, is a constant. It is not hard to see by definition, for any small $\epsilon > 0$, there is an n_0 such that, for every $n \geq n_0$, and every signature (θ, s),

$$\frac{\log S_T(L^{\text{simple}}_{(\theta,s)})}{T} < \lambda^* + \epsilon, \tag{3}$$

where $T = 2|(\theta,s)|n$, and among which, there are infinitely n satisfying

$$\frac{\log S_T(L^{\text{simple}}_{(\theta^*,s^*)})}{T} < \lambda^* - \epsilon, \tag{4}$$

where $T = 2|(\theta^*,s^*)|n$.

Now, we are ready to prove the theorem. We use $R_{\text{fntcrs}}(M)$ to denote the accepting runs of M that makes at most m (the number of states in M) sweeps. Notice that $R_{\text{fntcrs}}(M) \subseteq R_g(M)$ by definition, and $\lambda_{R_{\text{fntcrs}}(M)}$ is computable, according to Theorem 2. We now consider $R_{g,>m}(M)$ that is the set of all g-crossing-bounded accepting runs of M, each of which makes more than m sweeps.

Consider again $\alpha \in R_{g,>m}(M)$ be an accepting run on unary input word w (the w is encoded in the run). We use $n > n_0$ to denote the length of unary w and use $k = T/2n$ to denote the number of sweeps, where T is the length of α. As in before, we use $s^0 \cdots s^k$ to denote the state sequence of α. We are interested in estimating the number of accepting runs in $R_{g,>m}(M)$ with length T, and with the same state sequence. Clearly, one can delete all the simple loops from α, one by one. The result, which is not necessarily unique, is still an accepting run containing $k_0 \leq m$ sweeps. The deleted simple loops, totally containing $k - k_0$ sweeps, have combined length being exactly $2n(k - k_0)$. For each such simple loop, say a simple-s-loop, how many simple-s-loops can we use to replace the simple-s-loop in the original α such that the resulting α is still an accepting run? Using (3), the number is upper bounded by $2^{t(\lambda^*+\epsilon)}$, where t is the length of the simple-s-loop. From this, the total number of all possible accepting (n,k)-runs with the state sequence $s^0 \cdots s^k$, is bounded above by:

$$2^{2n(k-k_0)(\lambda^*+\epsilon)} \cdot 2^{2nk_0\lambda_0},$$

where the λ_0 is $2\log m + \log|\Sigma|$ is the maximum bit rate needed to code an instruction. Considering that there are at most $2^{k \log m}$ state sequences, the total number $S_{(n,k)}$ of all possible accepting (n,k)-runs satisfies,

$$\frac{\log S_{(n,k)}}{T} \leq \frac{k \log m + 2n(k-k_0)(\lambda^*+\epsilon) + 2nk_0\lambda_0}{T}, \tag{5}$$

where $T = 2nk$. Sending T to infinity, and using the fact that M is g-crossing-bounded (so n goes to infinity as well), we have

$$\lambda_{R_{g,>m}(M)} \leq \lambda^* + \epsilon. \tag{6}$$

Taking $\epsilon \to 0$, we therefore have,

$$\lambda_{R_{g,>m}(M)} \leq \lambda^*. \tag{7}$$

On the other hand, we consider the signature (θ^*, s^*). By definition, every simple-s^*-loop in $L^{\text{simple}}_{(\theta^*,s^*)}$ is part of an accepting run α that has signature (θ^*, s^*) and contains exactly one loop that is the simple-s^*-loop. Once this loop is dropped, the number of sweeps k_0 is uniquely specified in the signature θ^* (the length of η^* in θ^*). There are a monotonic and infinite sequence of numbers

$$n_1, \cdots, n_i, \cdots$$

with each satisfying (4) and $\lim n_i = \infty$. Define, accordingly, N_i to be the maximum number satisfying $|(\theta^*, s^*)|N_i < g(n_i) - m$. That is, the α is still a g-crossing-bounded accepting run with signature (θ^*, s^*) when the simple-s^*-loop is looped for N_i times. Notice that, by the definition of g, $N_i \to \infty$ as $i \to \infty$. Notice that every time it is looped, another simple-s^*-loop working on the same unary input word w can be chosen (we shall emphasize that, because w is unary, the length of input decides uniquely the w; this is why the proof does not work for non-unary 2NFA). Define $T_i = k_0 + 2|(\theta^*, s^*)|N_i n_i$, which is the length of the α with additional loops added. Using (4), this gives

$$\log S_{T_i}(R_{g,>m}(M)) \geq 2|(\theta^*, s^*)|N_i n_i(\lambda^* - \epsilon).$$

Immediately, taking $i \to \infty$, we have

$$\lambda_{R_{g,>m}(M)} \geq \lambda^* - \epsilon. \tag{8}$$

Again, sending $\epsilon \to 0$ and combining (7), we have $\lambda_{R_{g,>m}(M)} = \lambda^*$. The result follows, since $\lambda_{R_g(M)} = \max(\lambda^*, \lambda_{R_{\text{fntcrs}}(M)})$. □

It seems that removing the bounded-crossing condition from Theorem 3 is difficult. It relies on the following conjecture which we believe is likely true. Let L be a language. We use

$$\Lambda_L = \sup \frac{\log S_n(L)}{n}$$

to denote the max-rate of L. We conjecture that the max-rate of a regular language is computable. If this conjecture holds, we believe we can show the bounded-crossing condition can be removed from Theorem 3.

Clearly, for a two-way deterministic finite-automaton (2DFA), the execution information rate is computable, since, when it deterministic, it is finite-turn.

Now we consider 2-tape NFAs. In a 2-tape NFA, each instruction is in the following form: $(p, a, b) \to q$ where a, b can be ϵ. One can easily construct a DFA that accepts all the accepting runs of a 2-tape NFA. Therefore, the execution information of a 2-tape NFA is computable. The result can be generalized to multi-tapes.

3 Execution Information Rate of Some Classes of Infinite State Automata

A counter is a nonnegative integer variable that can be incremented by 1, decremented by 1, or stay unchanged. In addition, a counter can be tested against 0. A nondeterministic counter machine (NCM) is a one-way nondeterministic finite automaton augmented with k counters, for some k. An NCM M is reversal-bounded [13] if, for some nonnegative integer r, each counter is r-reversal-bounded; i.e., it makes at most r alternations between nondecreasing and nonincreasing modes in any computation. When M is deterministic, it is called a reversal-bounded DCM. As usual, $L(M)$ denotes the language that M accepts.

Reversal-bounded NCMs and their variations have been extensively studied since its introduction in 1978 [13]. In particular, reversal-bounded NCMs have found applications in areas like Alur and Dill's [1] time-automata [8,9], Paun's [16] membrane computing systems [14], and Diophantine equations [20]. For the purpose of this paper, the following result is fundamental:

Theorem 4. *The information rate of the language accepted by a reversal-bounded DCM is computable [7].*

Given a run α of a reversal-bounded NCM M, one can straightforwardly construct a reversal-bounded DCM to check whether α is an accepting run of M, where the reversal-bounded counters in M are directly simulated in the reversal-bounded DCM. We still use $R(M)$ to denote the set of all accepting runs of M. Hence, immediately from Theorem 4, the information rate of $R(M)$ is computable.

Theorem 5. *The execution information rate of a reversal-bounded NCM is computable.*

Similar to Theorem 2, one can generalize Theorem 5 to the case when the reversal-bounded NCM is finite-crossing.

Theorem 6. *The execution information rate of a finite-crossing reversal-bounded 2NCM is computable.*

Clearly, the result can be generalized to a reversal-bounded NCM with multi-tapes. We now consider a one-way nondeterministic pushdown automaton (PDA) M. Clearly, $R(M)$ can be accepted by a deterministic PDA; hence from the remarkable result by Kuich [15] showing that the information rate of an unambiguous context-free language is computable, we immediately have the information rate of $R(M)$ is also computable. This also generalizes for multi-tape nondeterministic PDA.

We now show the use of the results through examples. Consider a program M, consisting of two concurrent finite state processes M_1 and M_2 that receive signals from the environment. We can conveniently treat M_i $(i = 1, 2)$ as an NFA with alphabet Σ_i, while an input word w say, aba, is thought of a sequence of three "signals" fed into the process. In particular, we call $\Sigma_{12} = \Sigma_1 \cap \Sigma_2$ as the alphabet of synchronizing symbols. That is, when such a symbol a is fed into the concurrent program, both M_1 and M_2 will read the symbol (i.e., two "read a" instructions in M_1 and M_2 respectively run simultaneously). Notice that such a concurrent program is essentially a 2-tape NFA. Thus, the execution complexity of the concurrent M can be computed.

When M_1 and M_2 run concurrently, the speed of the environment proving the signals is not pre-defined. In other words, suppose that when the environment feeds a synchronized symbol to M, both M_1 and M_2 will read the symbol immediately without any delay. In case of an unsynchronized symbol fed to, say, M_1, M_1 will read immediately while M_2 will wait (since the symbol is not in its interface or Σ_2). In this understanding, a run of M depends on the delays between symbols fed by the environment, if we consider the run to be of discrete time. A popular model of real-time systems is timed automata [1], which are able to "generate" timed words. When discrete time is considered in a timed automaton, all such timed words form a regular language [1], where a special 'tick' symbol is used to indicate every time progress. In this setting, when both M_1 and M_2 are also synchronized on the tick symbols in a timed word, the M is essentially a 2-tape NFA. Hence the execution information rate of M can also be computed.

We now turn back to the untimed case. Suppose that all the signal sequences fed to M are drawn from a regular language and further satisfying a linear constraint P on the counts of signal symbols, say, the numbers of symbols a,b and c are all the same. The linear constraint can be used to specify a fairness constraint. In this case, it is not hard to see that M is essentially a two-tape NFA augmented with reversal-bounded counters, and hence the execution information rate of M can also be computed.

4 Experiments

In this section, we will evaluate the feasibility of computing execution information rates of real-world C programs. In order to do this, theoretically, we need to translate a C program into an infinite state automaton or a finite automaton with an astronomical number of states and then compute the execution information rate of the automaton. Indeed, it is an impractical approach (if not possible) since the computation may be inefficient, and even be uncomputable.

We take a more practical approach by using control flow graphs (CFGs), instead of automata, as an approximation model of executions of C programs. The benefits of using the CFG model are two fold: (1) translation from a C program to a CFG is efficient and frequently used in practice, which only need limited resources; (2) computing the information rate of a CFG is known efficient. Although, in previous sections, we proved that the execution information rates for some classes of finite automata and infinite state automata are computable, it does not imply that there is an efficient algorithm to calculate the execution information rate of them. However, the execution information rate of a CFG is much simpler: an "execution" on a CFG is viewed as a path on the CFG so that the information rate of the CFG is its execution information rate. Now, the CFG is simply treated as a graph whose information rate can be calculated efficiently theoretically [5] and practically [21].

Table 1. Yahtzee Game

No.	Time(sec)	Rate	Lines
1	30.49313463	0.28015439	223
2	21.27258946	0.29491658	211
3	1.59721724	0.09006925	56
4	1.41296586	0.20145839	87
5	6.28271311	0.16978757	212
6	1.33529892	0.09409675	96
7	310.29902566	0.24228938	536
8	0.48937783	0.23247921	69
9	25.09538204	0.12762918	222
10	5.68259796	0.20776446	170
11	1884.03881786	0.22348663	604
12	266.95954819	0.27088418	480
13	1.38500821	0.24429944	73
14	4.76470822	0.21707617	170
15	131.62702442	0.2388927	356
16	95.57444331	0.26930851	342
17	1.82504375	0.05810482	116
18	32.63405040	0.26382353	225
19	2.69637575	0.16071193	86
20	7.97615246	0.208024	146

Table 2. Battleship Game

No.	Time(sec)	Rate	Lines
1	3.25146617	0.14039641	72
2	4.04993043	0.11438705	105
3	4.91766219	0.2126072	117
4	4.50877264	0.2126072	137
5	1.38216377	0.10215237	135
6	6.39848402	0.2126072	131
7	187.69931130	0.11751557	291
8	5.59665177	0.2126072	123
9	24.34702401	0.13267324	199
10	5.78696373	0.2126072	143
11	37.79719132	0.13323103	208
12	8.62281422	0.15347345	170
13	2.59683185	0.05619995	129
14	10.42010990	0.12654569	198
15	241.68989470	0.03471603	418
16	31.62538405	0.11962904	376
17	6.11469088	0.07143966	103
18	6.06059559	0.2126072	149
19	10.47277698	0.23918791	192
20	14.35727940	0.22070435	170

Next, we will show the results of experiments on two sets of C programs to validate the feasibility of estimating execution information rates of real-world C programs: one set is twenty programs of Yahtzee game (a student programming project) and the other set is twenty programs from Battleship game (also a student program project).

Our procedure of doing the experiments is presented as follows:

- We run CoFlo, a open-source CFG generator, to translate a C program into a CFG on a Linux machine.
- Using a Python script, we convert the CFG into a matrix and generate a Matlab program which is able to perform eigenvalue calculation on the matrix.
- We run the Matlab program to calculate the information rate of CFG, which is the logarithm of the maximal eigenvalue of the matrix [5].

Note that, due to the capabilities of CoFlo, we can only generate CFG of a function, rather than the entire program. Hence, we use the CFG of the main function in a program to approximate that of the entire program. Although this approximation is lossy, it is still meaningful since, for students' programming projects, most control structures are contained in main functions of programs.

Experimental results are summarized in Table 1 and 2. In these tables, the term "Time(sec)" represents the execution time of the Matlab program calculating the execution information rate of a program. "Rate" is the value of execution information rate of a program. "Lines" represents the number of lines of the main function in a program. From these results, we conclude that execution information rate of programs with hundreds of lines is calculated in hours. How about the program with thousands of lines? In our experiments, there was an example, which is not shown in above tables, which is a program with almost 9000 lines of code. After several hours' execution, Matlab failed to calculate the execution information rate of the 9000 lines program because of an "out of memory" error. This implies, even applying an approximation using CFG, it is still difficult to compute the execution information rate of a program when the program is large. Therefore, more efficient approaches to calculate/estimate the execution information rates of large C programs need be investigated in future.

Our experiments run on a laptop with Intel P8600 processor and 4 GB memory using a student version Matlab software.

5 Conclusions

We studied the execution information rate of accepting runs of various forms of automata, including NFAs, finite-crossing 2NFAs, and a restricted form of unary 2NFA, reversal-bounded nondeterministic counter machines and their variations. We showed that in these cases, the execution information rates are computable. On the other hand, computing the rate for a two-way automaton is in general difficult because accepting runs for such an automaton are not necessarily regular nor context-free. We still need a better set of theoretical tools to handle such an automaton. In the practical side, we conducted experiments on C programs showing that computing/estimating information rates for their executions is feasible in many cases.

Acknowledgement. The authors would like to thank the anonymous reviewers for many constructive suggestions.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Asarin, E., Degorre, A.: Volume and Entropy of Regular Timed Languages: Discretization Approach. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 69–83. Springer, Heidelberg (2009)
3. Chhabra, J.K., Aggarwal, K.K., Singh, Y.: Code and data spatial complexity: two important software understandability measures. Information and Software Technology 45(8), 539–546 (2003)

4. Chhabra, J.K., Gupta, V.: Evaluation of object-oriented spatial complexity measures. SIGSOFT Softw. Eng. Notes 34(3), 1–5 (2009)
5. Chomsky, N., Miller, G.A.: Finite state languages. Information and Control 1, 91–112 (1958)
6. Cover, T.M., Thomas, J.A.: Elements of information theory, 2nd edn. Wiley-Interscience (2006)
7. Cui, C., Dang, Z., Ibarra, O.H., Fischer, T.R.: Information rate of some classes of non-regular languages: An automata-theoretic approach (2012) (submitted)
8. Dang, Z., Ibarra, O.H., Bultan, T., Kemmerer, R.A., Su, J.: Binary Reachability Analysis of Discrete Pushdown Timed Automata. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 69–84. Springer, Heidelberg (2000)
9. Dang, Z.: Pushdown timed automata: a binary reachability characterization and safety verification. Theor. Comput. Sci. 302(1-3), 93–121 (2003)
10. Douce, C.R., Layzell, P.J., Buckley, J.: Spatial measures of software complexity. In: Proc. 11th Meeting of Psychology of Programming Interest Group, Leeds (1999)
11. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc. (1977)
12. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley (2007)
13. Ibarra, O.H.: Reversal-bounded multicounter machines and their decision problems. Journal of the ACM 25(1), 116–133 (1978)
14. Ibarra, O.H., Dang, Z., Egecioglu, O., Saxena, G.: Characterizations of Catalytic Membrane Computing Systems. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 480–489. Springer, Heidelberg (2003)
15. Kuich, W.: On the entropy of context-free languages. Information and Control 16(2), 173–200 (1970)
16. Paun, G.: Membrane Computing, An Introduction. Springer (2002)
17. Shannon, C.E., Weaver, W.: The Mathematical Theory of Communication. University of Illinois Press (1949)
18. Shao, J., Wang, Y.: A new measure of software complexity based on cognitive weight. Can. J. Elect. Comput. Eng. 28(2), 69–74 (2003)
19. Staiger, L.: The Entropy of Lukasiewicz-Languages. In: Kuich, W., Rozenberg, G., Salomaa, A. (eds.) DLT 2001. LNCS, vol. 2295, pp. 155–165. Springer, Heidelberg (2002)
20. Xie, G., Dang, Z., Ibarra, O.H.: A Solvable Class of Quadratic Diophantine Equations with Applications to Verification of Infinite State Systems. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 668–680. Springer, Heidelberg (2003)
21. Yang, L., Cui, C., Dang, Z., Fischer, T.R.: An information-theoretic complexity metric for labeled graphs (2011) (in review)

Decidability and Complexity Results for Verification of Asynchronous Broadcast Networks

Giorgio Delzanno and Riccardo Traverso

DIBRIS, Università di Genova
via Dodecaneso 35, 16146 Genova, Italy
{Giorgio.Delzanno,Riccardo.Traverso}@unige.it

Abstract. We study decidability and complexity of verification problems for networks in which nodes communicate via asynchronous broadcast messages. This type of communication is achieved by using a distributed model in which nodes have a local buffer. We consider here safety properties expressed as a coverability problem with an arbitrary initial configuration. This formulation naturally models the search of an initial topology that may lead to an error state in the protocol.

Keywords: Infinite-state Systems, Broadcast Protocols, Verification.

1 Introduction

We present (un)decidability and complexity results for the coverability problem of Asynchronous Broadcast Networks (ABN), a mathematical model of distributed systems in which processes interact via topology-dependent and asynchronous communication. Our formal model of asynchronous broadcast communication combines three main features: a graph representation of a network configuration decoupled from the specification of individual process behaviour, a topology-dependent semantics of synchronization, and the use of local mailboxes to deliver messages to individual nodes. Our main abstraction comes from considering protocols defined via a communicating finite-state automaton replicated on each node of the network.

In our setting the coverability problem is formulated as follows. We first define an initial configuration as any graph in which nodes have labels that represent the initial state of the protocol (and no constraints on edges). Coverability consists then in checking whether there exists an initial configuration that can reach a target configuration that contains a given process state. A similar decision problem is considered in [8] for a mathematical model with synchronous communication and dynamic reconfiguration of the topology called Reconfigurable Broadcast Network (RBN).

Our analysis is carried out with different policies to handle buffers, namely unordered bags (an abstraction of a tuple space), and perfect or lossy FIFO channels. Our technical contribution is as follows. We first show that, in contrast with the synchronous case discussed in [9,10], coverability is decidable when local buffers are unordered. For the proof, we first give a reduction to the restricted case of fully connected topologies. We then solve the coverability problem through a reduction to the Cardinality Reachability Problem for Reconfigurable Broadcast Networks, a PTIME-complete problem [8]. The resulting algorithm is based on a forward labelling procedure described in detail in [8].

When mailboxes are ordered buffers, we obtain undecidability already in the case of fully connected topologies. The undecidability proof is based on a non-trivial encoding of the set of operations of a two counter machine in form of a cooperation protocol between distinct nodes. The protocol consists of different phases, each one is defined over a distinct set of control messages. The difficulty of the encoding comes from the fact that it is not possible to infer well-formedness properties for the content of the mailbox of an individual node. Thus it is not possible to encode the current value of the counters using the current content of a set of mailboxes. The current value of the counters is represented however in the flow of messages consumed by a pair of nodes elected in a preliminary phase of the protocol, which, in turn, completes successfully only under certain conditions on the sequence of consumed control messages. The coverability problem is decidable when introducing non-deterministic message losses. The results again follows from a reduction to the Cardinality Reachability Problem for RBN.

In an extended model in which a node can test if its mailbox is empty, we obtain undecidability with unordered bags and both arbitrary or fully-connected topologies. For this reduction we need to control the interferences due to the simultaneous communication with several neighbours. We exploit here the emptiness test in order to enforce the well-formedness of the mailboxes of nodes involved in the simulation of counter machines.

To our knowledge, the present work shows the first complexity analysis for (parameterized) coverability in formal models of asynchronous broadcast communication.

Detailed proofs and encodings are presented in the technical report [13].

2 Asynchronous Broadcast Network (ABN)

In this section we formally define our asynchronous model for broadcast communication. A configuration is defined as a labelled graph. Nodes correspond to processes running a common, pre-defined protocol. Each node has a local message buffer used to collect messages sent by neighbours.

A protocol is specified via a finite-state automaton with send and receive operations that correspond to write [resp. read] on remote [resp. local] buffers. Communication is topology-dependent, anonymous and asynchronous: when a process at node n sends a message a, the process does not block, and the message is added to the local mailbox of all of its neighbours without explicit information about the sender (i.e. messages do not contain node identifiers).

Formally, we consider a finite set Σ of messages, and different disciplines for handling the mailbox (message buffer), e.g., unordered mailboxes that we represent as bags over Σ, and ordered mailboxes that we represent as words over Σ.

In order to deal in a uniform way with different mailbox types we define a transition system parametric on the data structures used to model mailboxes. More specifically, we consider a mailbox structure $\mathbb{M} = \langle \mathcal{M}, del?, add, del, [] \rangle$, where $\mathcal{M}$ is a denumerable set of elements denoting possible mailbox contents; for $a \in \Sigma$ and $m \in \mathcal{M}$, $add(a, m)$ denotes the mailbox obtained by adding a to m, $del?(a, m)$ is true if a can be removed from m; $del(a, m)$ denotes the mailbox obtained by removing a from m when possible, undefined otherwise. Finally, $[] \in \mathcal{M}$ denotes the empty mailbox. We call an element a of m *visible* when $del?(a, m) = true$. Their specific semantics and corresponding properties change with the type of mailbox considered.

Definition 1. *A protocol is defined by a process $\mathcal{P} = \langle Q, \Sigma, R, q_0 \rangle$, where Q is a finite set of control states, Σ is a finite message alphabet, $Act = \{\tau\} \cup \{!!a, ??a \mid a \in \Sigma\}$, $R \subseteq Q \times Act \times Q$ is a set of transition rules, $q_0 \in Q$ is an initial control state.*

The label τ represents the capability of performing an internal action, and the label $!!a$ [$??a$] represents the capability of broadcasting [receiving] a message $a \in \Sigma$.

Definition 2. *Configurations are undirected $(Q \times \mathcal{M})$-graphs. A $(Q \times \mathcal{M})$-graph γ is a tuple $\langle V, E, L \rangle$, where V is a finite set of nodes, $E \subseteq V \times V$ is a finite set of edges (such that E is symmetric and $\forall v \in V.(v, v) \notin E$), and $L : V \to (Q \times \mathcal{M})$ is a labelling function.*

In the rest of the paper, for an edge $\langle u, v \rangle$ in E, we use the notation $u \sim_\gamma v$ and say that the vertices u and v are adjacent to one another in γ. We omit γ, and simply write $u \sim v$, when it is made clear by the context. We use $L(\gamma)$ to represent the set of labels in γ. The set of all configurations is denoted Γ, while $\Gamma_0 \subseteq \Gamma$ is the set of all initial configurations, in which nodes always have the same label $\langle q_0, [] \rangle$.

Given the labelling L and the node v s.t. $L(v) = \langle q, m \rangle$, we define $L_s(v) = q$ (state component of $L(v)$) and $L_b(v) = m$ (buffer component of $L(v)$). Furthermore, for $\gamma = \langle V, E, L \rangle \in \Gamma$, we use $L_s(\gamma)$ to denote the set $\{L_s(v) \mid v \in V\}$.

Definition 3. *For $\mathbb{M} = \langle \mathcal{M}, del?, add, del, [] \rangle$, an Asynchronous Broadcast Network (ABN) associated to $\mathcal{P}$ is a tuple $\mathcal{T}(\mathcal{P}, \mathbb{M}) = \langle \Gamma, \Rightarrow_{\mathbb{M}}, \Gamma_0 \rangle$, where $\Rightarrow_{\mathbb{M}} \subseteq \Gamma \times \Gamma$ is the transition relation defined next. For $\gamma = \langle V, E, L \rangle$ and $\gamma' = \langle V, E, L' \rangle$, $\gamma \Rightarrow_{\mathbb{M}} \gamma'$ holds iff one of the following conditions on L and L' holds:*

Local. *There exists $v \in V$ such that $(L_s(v), \tau, L'_s(v)) \in R$, $L_b(v) = L'_b(v)$, and $L(u) = L'(u)$ for each $u \in V \setminus \{v\}$.*

Broadcast. *There exists $v \in V$ and $a \in \Sigma$ such that $(L_s(v), !!a, L'_s(v)) \in R$, $L_b(v) = L'_b(v)$ and for every $u \in V \setminus \{v\}$*
- *if $u \sim v$ then $L'_b(u) = add(a, L_b(u))$ and $L_s(u) = L'_s(u)$,*
- *otherwise $L(u) = L'(u)$.*

Receive. *There exists $v \in V$ and $a \in \Sigma$ such that $(L_s(v), ??a, L'_s(v)) \in R$, $del?(a, L_b(v))$ is satisfied, $L'_b(v) = del(a, L_b(v))$, and $L(u) = L'(u)$ for each $u \in V \setminus \{v\}$.*

A local transition only affects the state of the process that executes it, while a broadcast also adds the corresponding message to the mailboxes of all the neighbours of the sender. Notice that broadcast is never blocking for the sender. Receivers can read the message in different instants. This models asynchronous communication. A reception of a message a is blocking for the receiver whenever the buffer is empty or the visible elements are all different from a. If a is visible in the mailbox, the message is removed and the process moves to the next state. Furthermore, it is easy to show that, when needed, a set $Q_0 \subseteq Q$ of initial states for $\mathcal{P}$ can be modelled by introducing a fresh initial state with outgoing local transitions to each $q \in Q_0$.

An *execution* is a sequence $\gamma_0\gamma_1\ldots$ such that γ_0 is an initial configuration, and $\gamma_i \Rightarrow_{\mathbb{M}} \gamma_{i+1}$ for $i \geq 0$. We use $\Rightarrow^*_{\mathbb{M}}$ to denote the reflexive and transitive closure of $\Rightarrow_{\mathbb{M}}$. We drop $\mathbb{M}$ when the mailbox type is clear from the context.

Decision Problem. The *Coverability Problem* parametric on the mailbox structure $\mathbb{M}$, abbreviated as $COV(\mathbb{M})$, is defined as follows.

Definition 4. *Given a protocol $\mathcal{P}$ with transition system $\mathcal{T}(\mathcal{P}, \mathbb{M}) = \langle \Gamma, \Rightarrow_{\mathbb{M}}, \Gamma_0 \rangle$ and a control state q, the coverability problem $COV(\mathbb{M})$ states: are there two configurations $\gamma_0 \in \Gamma_0$ and $\gamma_1 \in \Gamma$ such that $\gamma_0 \Rightarrow^*_{\mathbb{M}} \gamma_1$ and $q \in L_s(\gamma_1)$?*

In other words we require that a graph γ_q with a singleton node labelled q covers a reachable configuration γ_1, i.e., γ_q is a subgraph of γ_1. We often use the terminology γ_0 reaches state q as an abbreviation for $\gamma_0 \Rightarrow^*_{\mathbb{M}} \gamma_1$ and $q \in L_s(\gamma_1)$ for some configuration γ_1. Besides being parametric on the mailbox structure, our decision problem is parametric on the shape of the initial configuration. As mentioned in the introduction, this feature models in a natural way verification problems for protocols with partial information about the structure of the network.

2.1 ABN vs RBN

In the rest of the paper we will often refer to the semantics of RBN models [8]. Protocols in RBN adhere to the same syntax as ABN. Configurations are simply Q-graphs, i.e., graphs in which nodes have labels in Q via the labelling function L. The semantics of broadcast communication however is synchronous instead of asynchronous. Furthermore, the topology of the network may nondeterministically change. Formally, given $R_a(q) = \{q' \in Q \mid \langle q, ??a, q' \rangle \in R\}$ and two Q-graphs θ, θ' with $\theta = \langle V, E, L \rangle$, we have $\theta \to \theta'$ iff $\theta' = \langle V, E', L' \rangle$ and one of the following conditions holds:

Synch Broadcast. $E' = E$ and $\exists v \in V$ s.t. $\langle L(v), !!a, L'(v) \rangle \in R$ and $L'(u) \in R_a(L(u))$ for every $u \sim v$, and $L(w) = L'(w)$ for any other node w.
Reconfiguration. $E' \subseteq V \times V \setminus \{\langle v, v \rangle \mid v \in V\}$ and $L = L'$.

3 Unordered Mailboxes

In this section we study the coverability problems for ABNs in which mailboxes are unordered buffers modelled as bags over the finite message alphabet Σ. The mailbox structure *Bag* is defined as follows: $\mathcal{M}$ is the denumerable set of bags over Σ, $add(a, m) = [a] \oplus m$ (multiset sum of the singleton $[a]$ and m), $del?(a, m) = true$ iff $m(a) > 0$, $del(a, m) = m \ominus [a]$ (multiset removal of $[a]$ from m), and $[] \in \mathcal{M}$ is the empty bag $[]$. The operational semantics follows from the general definitions.

Let us consider the instance $COV(Bag)$ of the coverability problem. For synchronous broadcast, coverability is undecidable for arbitrary topologies [9]. We show next that coverability is in PTime for unordered mailboxes.

For the ease of notation, we use $\mathcal{T}^{\mathcal{K}}(\mathcal{P}, \mathbb{M})$ [resp. $COV_{fc}(\mathbb{M})$] to denote the restriction of $\mathcal{T}(\mathcal{P}, \mathbb{M})$ [resp. $COV(\mathbb{M})$] to fully connected configurations only, i.e., configurations such that $u \sim_\gamma v$ for each pair of distinct nodes $u, v \in V$. We prove the results in two different steps. We first show that, for the purpose of deciding $COV(Bag)$, we can focus on fully connected topologies only. We then show a reduction from $COV_{fc}(Bag)$ to the Cardinality Reachability Problem for Reconfigurable Broadcast Networks, that, for short, we will refer to as CRP. The reduction requires reachability queries of the form $\#q \geq 1$ (at least one occurrence of control state q). The latter problem is PTime-complete [8].

For asynchronous communication with unordered mailboxes, coverability for arbitrary topologies case can be reduced to the fully connected case. The following lemma indeed holds.

Lemma 5. *Given an ABN protocol $\mathcal{P} = \langle Q, \Sigma, R, q_0 \rangle$ and a state $q \in Q$, if there exists an arbitrary topology from which we can reach state q, then there exists a fully connected topology from which we can also reach q.*

One side of the property is immediate. If there exists a fully connected initial configuration that reaches a configuration in which state q occurs, then coverability is solved. In order to prove the other implication, the intuition is that we can exploit the fact that mailboxes are unordered to ignore messages sent along links that are not present in a given topology.

The following lemma relates coverability in ABN to the cardinality reachability problem in RBN.

Lemma 6. *Given an ABN protocol $\mathcal{P} = \langle Q, \Sigma, R, q_0 \rangle$ and a state $q \in Q$ let $\mathcal{P}'$ be the RBN protocol with the same rules but with $\{q_0\}$ as singleton set of initial states. Then, there exists an execution of $\mathcal{P}'$ that satisfies CRP if and only if there exists an execution of $\mathcal{P}$ satisfying $COV_{fc}(Bag)$.*

In the proof we can delay message receptions to simulate deletions of links. Vice versa, we can exploit reconfigurations and the possibility of adding nodes to the initial configuration to simulate asynchronous receipts using dynamically created links and synchronous messages. The previous reduction is done in constant time, since there is no need of modifying the protocol specification. We can therefore conclude that the following property holds.

Theorem 7. *$COV(Bag)$ is* PTIME-*complete.*

Proof. Thanks to Lemmas 5 and 6 and to the PTIME algorithm for coverability in RBN [8], we know that $COV(Bag)$ is in PTIME. Completeness follows from a reduction of the Circuit Value Problem (CVP) [19] to $COV(Bag)$. Given an acyclic circuit G composed by a finite set of gates and a fixed evaluation of its inputs, CVP consists in evaluating G in the inputs. The reduction is based on a protocol in which a special node broadcasts the evaluation of a single input (in form of a message with label $true/false$ and an index associated to the corresponding variable). Gates (i.e. Boolean operations like and/or/not) are simulated by processes running on nodes. For each gate, we have nodes that receive the inputs, evaluate the gate, and broadcast their output to the other nodes. A special node intercepts the *true* message corresponding to the output of the whole circuit and moves in an acceptance state. Regardless the type of communication topology, number of nodes simulating each gate, and possible delays, coverability of the acceptance state corresponds to satisfiability of the circuit G w.r.t. the given assignment. □

4 FIFO Mailboxes

In this section we move to ABN with perfect FIFO buffers as communication media. In this context we instantiate the mailbox structure $FIFO$ as follows: $\mathcal{M}$ is defined as Σ^*; $add(a, m) = m \cdot a$ (concatenation of a and m); $del?(a, m) = true$ iff $m = a \cdot m'$; $del(a, m)$ is the string m' whenever $m = a \cdot m'$, undefined otherwise; finally, $[] \in \mathcal{M}$ is the empty string ϵ.

Theorem 8. *$COV(FIFO)$ and $COV_{fc}(FIFO)$ are undecidable.*

Proof. The proof is based on a reduction of the halting problem for two-counter machines – a well known undecidable problem – to $COV(FIFO)$. A two-counter machine is defined by a pair $\langle Loc, Inst \rangle$ where Loc is a finite set of control locations and $Inst \subseteq Loc \times Op \times Loc$ is a finite set of instructions such that $Op = \{c{+}{+}, c{-}{-}, c == 0 \mid c \in \{x_1, x_2\}\}$ is a set of operators over the counters x_1 and x_2, and $\ell_0 \in Loc$ is the initial location. Configurations are tuples $\langle \ell, v_1, v_2 \rangle$ such that $\ell \in L$ is the current location and v_1, v_2 are natural numbers that denote the current value of x_1 and x_2, respectively. The operational semantics is defined in a standard way: the execution of increment and decrement updates the control location and the current value of the corresponding counter, a zero-test updates the location whenever the test is satisfied in the current state of the counter.

The rationale behind the reduction of coverability to the halting problem of two-counter machines is as follows. We first use an election protocol that assigns fixed roles (controller/slave) to a pair of adjacent nodes. Since the initial configuration is not fixed a priori our election protocol does not forbid the election of multiple pairs of controller/slave nodes, but we only require that at least one pair is elected in order to succeed. The controller/slave nodes set up their mailboxes in order to use them as overlapping circular queues. Messages represent the current value (in unary) of the counters. The simulation is guided by the controller. The slave forwards all received messages back to the controller. As an example, to check that x_1 is zero, the controller reads all messages in the mailbox and checks that in between two successive reads of the marker for x_1 there are no units. We use interference to denote an unwanted message occurring in the mailbox of a controller/slave node. Since the network topology is not fixed a priori, a key point of the whole construction is the capability of controlling interferences with other nodes, e.g., avoiding the adjacency between multiple controllers and slaves. For this purpose, we use special control messages to coordinate the different phases and exploit the FIFO mailboxes in order to enforce the simulation to get into a deadlock state whenever the same control message is received more than once. A detailed description of the protocol is in [13]. The same construction can be used for the fully connected case. □

5 Lossy FIFO Mailboxes

We now consider coverability for ABNs in which mailboxes are lossy FIFO channels, i.e., channels in which messages may non-deterministically be lost. Given a protocol $\mathcal{P}$, a configuration γ of $\mathcal{T}^{\mathcal{K}}(\mathcal{P}, LFIFO)$ is a multiset of pairs $\langle q, m\rangle$ where $q \in Q$ and $m \in \Sigma^*$. To model non-deterministic loss of messages, we modify the operational semantics by introducing lossy steps.

We first need to define the ordering $\preccurlyeq$ between configurations. For $\gamma = \langle V, V \times V, L\rangle$ and $\gamma' = \langle V', V' \times V', L'\rangle$ $\gamma \preccurlyeq \gamma'$ iff there exists an injection $h : V \to V'$ s.t. $L_s(v) = L_s(h(v))$ and $L_b(v) \prec L_b(h(v))$ for each $v \in V$, where $\prec$ denotes the subword relation, namely, for $w, w' \in \Sigma^*$, $w \prec w'$ iff there exists an injective and strictly monotone mapping $h : |w| \to |w'|$ s.t. $w_i = w'_{h(i)}$ for $i : 1, \ldots, |w|$, where v_i denotes the i-th symbol in the word v. Intuitively, $\gamma \preccurlyeq \gamma'$ means that γ is obtained from γ' by removing nodes (and all corresponding edges) and messages from the buffers. We modify the transition relation $\Rightarrow$ to include lossy steps before and after each transition in the original system as follows: $\gamma \longmapsto \gamma'$ iff there exists η and ν s.t. $\eta \preccurlyeq \gamma$, $\eta \Rightarrow \nu$, and $\gamma' \preccurlyeq \nu$.

The ordering $\preccurlyeq$ is a simulation relation and is also a well-quasi ordering. These two properties pave the way for a possible application of the theory of well-structured transition systems [12] to solve coverability. In the rest of the section we use a reduction to RBN-coverability to obtain better complexity results. As for unordered mailbox we first show that we can focus our attention on fully connected topologies, only.

Lemma 9. *There exists an execution of $\mathcal{P}$ that satisfies $COV(LFIFO)$ if and only if there is one of $\mathcal{P}$ satisfying $COV_{fc}(LFIFO)$.*

We are now at the most tricky part of the proof that consists in proving that $COV_{fc}(LFIFO)$ can be reduced to CRP. Let $\mathcal{P}$ be an ABN protocol, and let $\mathcal{P}'$ be the corresponding RBN protocol derived as in Section 3.

Lemma 10. *There exists an execution of $\mathcal{P}'$ that satisfies CRP if and only if there is one of $\mathcal{P}$ satisfying $COV_{fc}(LFIFO)$.*

The coverability problem for lossy FIFO mailboxes has a property in common with the one for bags, that is in both cases processes are able to ignore incoming messages indefinitely; this is achieved by either leaving the message in the multiset or by deleting it from the lossy FIFO queue. We can therefore take again advantage of this property to obtain the following theorem.

Theorem 11. *$COV_{fc}(LFIFO)$ is* PTIME*-complete.*

Proof. Membership to PTIME follows from the reduction to CRP for RBNs. Hardness follows again from a reduction of CVP to COV for ABN lossy FIFO queues. The encoding protocol is the same as for unordered mailboxes. □

6 ABN with Emptiness Test

In this section we enrich the ABN model with a new type of transitions in order to enable nodes to test whether their mailbox is empty. We call the resulting model ABN$_\epsilon$. The set *Act* of action labels is extended to include ϵ, i.e., $Act = \{\tau, \epsilon\} \cup \{!!a, ??a \mid a \in \Sigma\}$. The transition systems associated to an ABN$_\epsilon$ are changed accordingly to take ϵ into account; given two configurations $\gamma = \langle V, E, L\rangle$ and $\gamma' = \langle V, E, L'\rangle$, $\gamma \Rightarrow \gamma'$ holds also if the following condition is met.

Emptiness Test. There exists a $v \in V$ such that $(L_s(v), \epsilon, L'_s(v)) \in R$, $L_b(v) = L'_b(v) = []$, and $L(u) = L'(u)$ for each $u \in V \setminus \{v\}$.

The only difference w.r.t. the semantics of τ-transitions consists in the $L_b(v) = []$ condition, that ensures that ϵ-transitions only fire when the mailbox is empty.

The introduction of ϵ-transitions affects the different instances of the coverability problem in different ways. The simplest case is for $COV_{fc}(FIFO)$ and $COV(FIFO)$, which of course are still undecidable: the possibility to test the emptiness of the mailbox does not have any effect on the reduction from two-counter machines. The reduction from $COV_{fc}(LFIFO)$ to CRP of Lemma 10 has to be modified in order to consider also ϵ-transitions. Given two configurations $\gamma, \gamma' \in \Gamma$ such that $\gamma \preccurlyeq \gamma'$ (see Section 5 for the definition of the $\preccurlyeq$ ordering), if ϵ is enabled in γ then it can be fired starting from γ' too, through a preliminary lossy step that empties the relevant mailbox. This means that ϵ-transitions are almost the same as internal transitions in case of $LFIFO$ mailboxes. Therefore, given a protocol $\mathcal{P} = \langle Q, \Sigma, R, q_0\rangle$ and a target state $q \in Q$, we derive an RBN protocol $\mathcal{P}' = \langle Q, \Sigma, R', \{q_0\}\rangle$ where R' is the set of rules R where all occurrences of ϵ have been replaced by τ, and then we solve CRP for the target state q. Thanks to the previously mentioned property of ϵ-transitions, one could adapt easily enough the proof of Lemma 10 to this case. From these observations we can therefore derive that both $COV(LFIFO)$ and $COV_{fc}(LFIFO)$ are decidable even with ϵ-transitions.

We incur in a completely different case when considering bags: as it can be shown, the extended semantics traces indeed a sharp boundary between decidability and undecidability. Without the emptiness test, both reachability problems $COV(Bag)$ and $COV_{fc}(Bag)$ are decidable; we prove that the operator ϵ

introduced with the extended model is sufficient to make them undecidable. The proof proceeds by building a reduction from the control state reachability problem for two-counter machines to $COV(Bag)$. The reduction encodes a counter machine $\mathcal{M}$ with an ABN protocol $\mathcal{P} = \langle Q, \Sigma, R, q_0 \rangle$ where, like before, each location $\ell \in Loc$ and each instruction $i \in Inst$ corresponds respectively to a state $\mathcal{P}(\ell) \in Q$ and to a set of intermediate states and rules. The protocol is split in two phases. In the first phase processes follow a distributed election protocol to identify who takes care of which role and who is excluded from the simulation. The second phase is the simulation of $\mathcal{M}$. The alphabet is partitioned in two sets, Σ_e for the election and Σ_s for the simulation. Since we do not make any particular assumption on the connectivity graph, the proof works for both $COV_{fc}(Bag)$ and $COV(Bag)$.

Election. A simulation must be carried out by three nodes: a controller and two slaves, one per counter. Figure 1 shows the protocol used to choose such roles. We say that a node is *in simulation* if it reaches (at least once) $\mathcal{P}(\ell_0)$, q_{S_1}, or q_{S_2}. The election guarantees minimal connectivity requirements, as stated in the following Lemma.

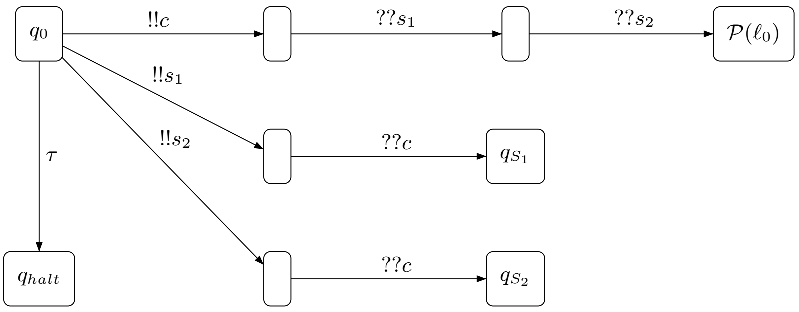


Fig. 1. $COV(Bag)$: Election protocol

Lemma 12. *If a node is in state $\mathcal{P}(\ell_0)$, then at least two of its neighbours are already respectively in state q_{S_1} and q_{S_2} or they can possibly move only those states. If a node is in state q_{S_1} or q_{S_2}, then at least one of its neighbours is already in state $\mathcal{P}(\ell_0)$ or it can possibly move only to $\mathcal{P}(\ell_0)$.*

Simulation. Each slave S_j keeps in its mailbox a number of u_j messages equal to the current value of counter x_j. The controller sends messages sub_j or tz_j to give orders depending on the instruction (ℓ, op, ℓ') that is going to be simulated by the system and waits for the slave which manages the involved counter to react accordingly (see Figure 2). Once the slave is done, the same control message is sent back to the controller as acknowledgement and the controller is able to proceed. When A is a set we write $??A$ to mean that for every $a \in A$ the protocol has a reception rule $??a$ with the same endpoints. Again, the increment can be done directly by the controller with a single broadcast $!!u_j$. In order to be able

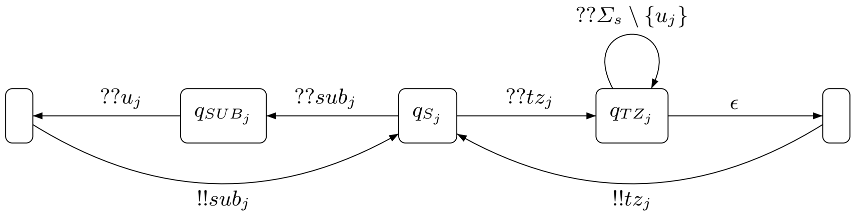


Fig. 2. $COV(Bag)$: Slave process

to prove the correctness of the reduction, we first state some properties of the simulation phase.

Lemma 13. *Any $m \in \Sigma_e$ received by a node in simulation will persist in its mailbox forever. Such a node is said to be* in interference.

Proof. By construction, for all $m \in \Sigma_e$, there are no receptions of m starting from any state which may be reached by simulating nodes. □

Lemma 14. *At any time, the value of the counter i is equal to the number of occurrences of units u_i in the mailbox of the corresponding slave, provided that no simulating node is in interference. We say then that the counters are* valid.

We remark that the notion of validity of the counters does not have anything to do with the compliance of their values w.r.t. the ones of the two-counter machine being simulated. Moreover, since the simulation may proceed even with invalid counters, the reduction does not compute reachability of the encoding $\mathcal{P}(\ell_f)$ of the target state ℓ_f, but instead it checks for the reachability of a fresh state q_{target} added according to Figure 3. This is needed in order to ensure the correctness of the simulation. It is straightforward to check that the instructions added to

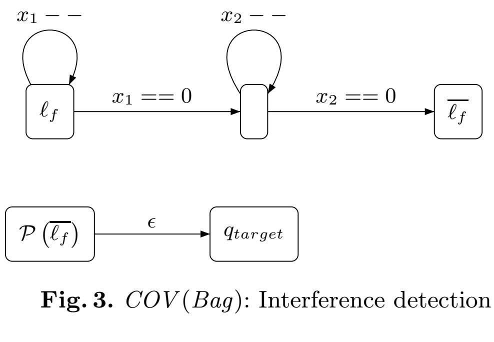


Fig. 3. $COV(Bag)$: Interference detection

$\mathcal{M}$ do not have any impact on the reachability of the target location, as they just decrement down to zero both counters before reaching the destination. We are now ready to prove that the reduction is indeed a correct simulation of the given two-counter machine.

Theorem 15. *$COV(Bag)$ [$COV_{fc}(Bag)$] is undecidable in ABN_ϵ.*

The correctness of the reduction can be proved by first demonstrating by induction on the number of simulated instructions that for any number of steps, either the counters will be valid and consistent w.r.t. the corresponding state of the two-counter machine or they will be (and remain) invalid. Given this property, we can exploit Lemma 13 in order to show that the added, final transitions from Figure 3 ensure that the controller will deadlock before reaching the target state when the counters are invalid. A detailed description of the protocol is in [13].

7 Related Work

Formal models of broadcast protocols in fully connected topologies have been defined in [14]. The model is based on extensions of Petri nets with whole place operations, used to model cache coherence protocols [6]. The coverability problem for broadcast protocols is decidable in fully connected graphs [16,7]. This problem is strictly related to marking coverability in Petri nets with transfer or reset arcs [1]. When individual processes are distributed over graphs of arbitrary shape, coverability becomes undecidable as shown in [9]. Decidability holds for special classes like bounded path graphs under the induced graph ordering [9,10], in presence of communication failures or interferences [11], and with dynamic reconfiguration of the communication topology [8]. The PTIME decision procedure in [8] is similar to the labelling algorithms used for parameterized verification of synchronous systems in [18]. In the timed case coverability becomes undecidable already in special types of star topologies [2].

Other formal models of broadcast communication have been proposed in [20,22,15,17]. Verification of unreliable communicating FIFO systems have been studied in [3,4]. In [5] the authors consider different classes of topologies with mixed lossy and perfect channels. The complexity of the verification procedures for lossy FIFO channel systems and broadcast protocols (transfer and reset nets) is discussed in [21]. A classification of the expressive power of different infinite-state models including lossy FIFO channel systems and broadcast protocols is discussed in [1].

Differently from all the previous works, we consider here coverability for parametric initial configurations for a distributed model with asynchronous broadcast. Furthermore, we also consider different policies to handle the message buffers as well as unreliability of the communication media. Finally, our new complexity results improve the preliminary analysis presented in the extended abstract [12], where we used well-structured transition systems for evaluating decidability for bags and lossy FIFO systems.

References

1. Abdulla, P.A., Delzanno, G., Begin, L.V.: A classification of the expressive power of well-structured transition systems. Inf. Comput. 209(3), 248–279 (2011)
2. Abdulla, P.A., Delzanno, G., Rezine, O., Sangnier, A., Traverso, R.: On the Verification of Timed Ad Hoc Networks. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 256–270. Springer, Heidelberg (2011)

3. Abdulla, P.A., Jonsson, B.: Undecidable verification problems for programs with unreliable channels. Inf. Comput. 130(1), 71–90 (1996)
4. Cécé, G., Finkel, A., Iyer, S.P.: Unreliable channels are easier to verify than perfect channels. Inf. Comput. 124(1), 20–31 (1996)
5. Chambart, P., Schnoebelen, P.: Mixing Lossy and Perfect Fifo Channels. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 340–355. Springer, Heidelberg (2008)
6. Delzanno, G.: Constraint-based verification of parameterized cache coherence protocols. FMSD 23(3), 257–301 (2003)
7. Delzanno, G., Esparza, J., Podelski, A.: Constraint-Based Analysis of Broadcast Protocols. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 50–66. Springer, Heidelberg (1999)
8. Delzanno, G., Sangnier, A., Traverso, R., Zavattaro, G.: On the complexity of parameterized reachability in reconfigurable broadcast networks. In: FSTTCS 2012, vol. 18, pp. 289–300. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2012)
9. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized Verification of Ad Hoc Networks. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
10. Delzanno, G., Sangnier, A., Zavattaro, G.: On the Power of Cliques in the Parameterized Verification of Ad Hoc Networks. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 441–455. Springer, Heidelberg (2011)
11. Delzanno, G., Sangnier, A., Zavattaro, G.: Verification of Ad Hoc Networks with Node and Communication Failures. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE 2012. LNCS, vol. 7273, pp. 235–250. Springer, Heidelberg (2012)
12. Delzanno, G., Traverso, R.: A formal model of asynchronous broadcast communication (preliminary results). In: ICTCS 2012 (2012), `http://ictcs.di.unimi.it/papers/paper_29.pdf`
13. Delzanno, G., Traverso, R.: On the coverability problem for asynchronous broadcast networks (extended and revised version). Tech. rep., TR-12-05, DIBRIS, University of Genova (November 2012), `http://verify.disi.unige.it/publications/`
14. Emerson, E.A., Namjoshi, K.S.: On model checking for non-deterministic infinite-state systems. In: LICS, pp. 70–80 (1998)
15. Ene, C., Muntean, T.: A broadcast-based calculus for communicating systems. In: IPDPS 2001, p. 149 (2001)
16. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999, pp. 352–359 (1999)
17. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: A Process Algebra for Wireless Mesh Networks. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 295–315. Springer, Heidelberg (2012)
18. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
19. Ladner, R.E.: The circuit value problem is log space complete for p. SIGACT News 7(1), 18–20 (1975)
20. Prasad, K.V.S.: A calculus of broadcasting systems. Sci. Comput. Program. 25(2-3), 285–327 (1995)
21. Schnoebelen, P.: Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 616–628. Springer, Heidelberg (2010)
22. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. 75(6), 440–469 (2010)

The Buffered π-Calculus: A Model for Concurrent Languages[⋆]

Xiaojie Deng[1], Yu Zhang[2], Yuxin Deng[1], and Farong Zhong[3]

[1] BASICS, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
[2] State Key Laboratory for Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3] Department of Computer Science, Zhejiang Normal University, Zhejiang, China

Abstract. Message-passing based concurrent languages are widely used in developing large distributed and coordination systems. This paper presents the buffered π-calculus — a variant of the π-calculus where channel names are classified into buffered and unbuffered: communication along buffered channels is asynchronous, and remains synchronous along unbuffered channels. We show that the buffered π-calculus can be fully simulated in the polyadic π-calculus with respect to strong bisimulation. In contrast to the π-calculus which is hard to use in practice, the new language enables easy and clear modeling of practical concurrent languages. We encode two real-world concurrent languages in the buffered π-calculus: the (core) Go language and the Core Erlang. Both encodings are fully abstract with respect to weak bisimulations.

Keywords: process calculus, formal model, full abstraction.

1 Introduction

Concurrent programming languages become popular in recent years thanks to the large demand of distributed computing and the pervasive exploitation of multi-processor architectures. Unlike the shared-memory concurrency model, which is now mainly used on multi-processor platforms, message passing based concurrent languages are particularly popular in developing large distributed, coordination systems. Indeed, quite a few real-world concurrent languages are intensively used in industry. The most well-known languages are probably Erlang, developed by Ericsson [1], and the much younger language Go, developed by Google [6]. Both languages achieve their asynchronous communication via order-preserving message passing.

On the other side, the π-calculus [11,14] has shown its success in modeling and verifying both specifications and implementations. Its asynchronous variant [3,8] is a good candidate as the target formal model. Despite the fact that it is

⋆ Partially supported by Natural Science Foundation of China (61173033, 61033002, 61161130530, 61100053) and the Opening Fund of Top Key Discipline of Computer Software and Theory in Zhejiang Provincial Colleges at Zhejiang Normal University.

called asynchronous, communication in the asynchronous π-calculus is however synchronous. It is shown in [2] that the communication modelled by the asynchronous π-calculus is equivalent to message passing via bags — senders put messages into some bags, and receivers may get arbitrary messages from these bags. This result indicates that additional effort should be made to respect the order of the messages, which is adopted in the implementation of many concurrent languages.

In view of this, we may expect a formal model where asynchronous communication is supported natively. In fact, our primary goal is to achieve a formal model by which we can easily define a formal semantics of Go and do verification on top of it. The developers of Go claim that the concurrency feature of Go is rooted in CSP [7], while we show that the π-calculus should be an appropriate model for Go as CSP does not support a channel passing mechanism.

In the spirit of the name passing mechanism of the π-calculus and the channel type of the Go language, we extend the π-calculus by introducing a special kind of names, each associated with a first-in-first-out buffer. We call these names *buffered names*. Communication along buffered names is asynchronous, while that along unbuffered (normal) names remains synchronous. We call this variant language the buffered π-calculus, and abbreviate it as the π_b-calculus.

We develop the π_b-calculus by defining its operational semantics as a labelled transition system and supplying an encoding into the polyadic π-calculus. We also present translations of the languages Go and Erlang into the π_b-calculus and show that the model is sufficient and relatively easier for modeling real-world concurrent languages.

Beauxis *et al* introduced the $\pi_{\mathfrak{B}}$-calculus in order to study the asynchronous nature of the asynchronous π-calculus [2]. Their asynchronous communication is achieved via explicit use of buffers. In case that the buffers are ordered structures such as queues or stacks, the asynchronous communication modelled by $\pi_{\mathfrak{B}}$ differs from that by the asynchronous π-calculus. While communication in the $\pi_{\mathfrak{B}}$-calculus is always asynchronous, we keep both synchronous and asynchronous communication in the π_b-calculus, through different types of names.

Encoding programming languages in process calculus have been studied by many researchers. Milner defines the semantics of a non-trivial parallel programming language by a translation into CCS in [9]. In [15], a translation from a parallel object oriented language to the minimal π-calculus is presented. The correctness of the translation is justified by the operational correspondence between units and their encodings. Our treatments to the Go language follows the approach in [15]. In addition, we show a full abstraction theorem, namely equivalent Go programs are translated into equivalent π_b processes.

For functional languages, Noll and Roy [12] presented an initial translation mapping from a Core Erlang [4] to the asynchronous π-calculus. Later on they [13] improved the translation by revising the non-deterministic encoding of pattern matching based expressions, and by adding the encoding for tuples. Their translations, however, are not sound in the sense that the order of messages is not always respected. By modelling the mailbox structure explicitly by buffered names in the π_b-calculus, we obtain a more accurate encoding which is fully abstract with respect to weak bisimulation.

The rest of the paper is structured as follows. Section 2 presents the syntax and semantics of the π_b-calculus and a simple encoding in the polyadic π-calculus [10]. We show that this encoding preserves the strong bisimulation relation. In Section 3 we define a formal semantics for Go and present an encoding of Go in the π_b-calculus. Due to page limit, the encoding of Erlang is supplied in the full version of the current paper [5], as well as many definitions and proofs. Finally, Section 4 concludes the paper.

2 The π_b-Calculus

We assume an infinite set $\mathcal{N}$ of names, ranged over by a, b, c, d, x, y. Processes are defined by the following grammar:

$$P, Q, \ldots := \textstyle\sum_{i \in I} \pi_i.P_i \;\Big|\; P|Q \;\Big|\; (\nu c : n)P \;\Big|\; (\nu c)P \;\Big|\; !P$$

where $\pi = c(x) \mid \overline{c}\langle d \rangle \mid \tau$.

Most of the syntax is standard: $\sum_{i \in I} \pi_i.P_i$ is the guarded choice (I is finite), which behaves nondeterministically as one of its components $\pi_j.P_j$ for some $j \in I$; composition $P|Q$ acts as P and Q running in parallel; $!P$ is the replication of process P; Prefixes $c(x)$ and $\overline{c}\langle d \rangle$ are input and output along name c; and τ is the silent action. We write $\mathbf{0}$ for the empty guarded choice, it is the process which can do nothing.

The π_b-calculus extends the π-calculus in the fact that names can be buffered or unbuffered. Unbuffered names are names in the π-calculus, and buffered names have the buffer attribute specified by a *buffer store*. A buffer store, denoted by $\mathcal{B}$, is a partial function from buffered names to pairs (n, l), where n is a positive integer representing the capacity of the buffer, and l is a list of names in the buffer, with the same order. Both $(\nu c)P$ and $(\nu c : n)P$ are called *new processes*. The (standard) new process $(\nu c)P$ specifies that c (whether buffered or unbuffered) is a local name in P. The extended new process $(\nu c : n)P$ creates a local buffered name c, whose associated buffer has the capacity n for asynchronous communication inside P. Notice that $(\nu c)P$ only says that the name c is local and does not imply that c is unbuffered — c can be a buffered name whose buffer is already created in the buffer store.

Input process $c(x).P$ and output process $\overline{c}\langle d \rangle.P$ can communicate with each other along name c when they run in parallel. If c is an unbuffered name, the communication is synchronous and happens as in the π-calculus: the object d is passed from the output side to the input side. If c is a buffered name, then the communication becomes asynchronous: the output process simply puts d into the buffer of c if it is not full and continues, or blocks if the buffer is full; the input process retrieves the oldest value from the buffer of c if it is not empty and continues, or blocks if the buffer is empty.

As usual, we write $\tilde{c}$ for a sequence of names, and abbreviate $(\nu c_1) \ldots (\nu c_n)P$ to $(\nu c_1 \ldots c_n)P$. A name x is bound if it appears in input prefix, otherwise it is free. We write $P\{\tilde{c}/\tilde{x}\}$ for the process resulting from simultaneously substituting

c_i for each free x_i in P. The newly created name c in $(\nu c : n)P$ or $(\nu c)P$ are local names. A name is global if it is not localized by any new operator. We use $ln(P)$ and $gn(P)$ for the set of local names and global names occurring in P.

Throughout the development of the paper, we assume the following *De Barendregt name convention*: *Local names are different from each other and from global names.* For instance, we shall never consider processes like $\overline{a}\langle c\rangle.(\nu a)P$ or $(\nu a)(\nu a)P$. We note that this convention is dispensable and we simply adopt it to make the presentation of the calculus simple and clean. One can also remove the convention and use syntactic rules to manage name conflicts, but dealing with names in buffers can be very subtle.

A process can send a local name into a buffer. The fact that a name stored in buffers is local must be tracked, because it may affect the name scope when another process retrieves this name from the buffer. The convention also works for buffer stores. We shall discuss more on this when defining the operational semantics. Inside a buffer store, a value of the form (νc) indicates that the name c was sent into the buffer when it was local. Given a buffer store $\mathcal{B}$, we write $gn(\mathcal{B}(b))$ for the set of global names that occur in b's buffer, and $gn(\mathcal{B}) = \bigcup_{b\in \mathtt{dom}(\mathcal{B})} gn(\mathcal{B}(b))$. Similarly $ln(\mathcal{B}(b))$ and $ln(\mathcal{B})$ for local names in $\mathcal{B}(b)$ and $\mathcal{B}$. The buffer store $\mathcal{B}\{c/d\}$ is obtained by substituting c for each d in $\mathcal{B}$.

We say a process Q is guarded in P, if every occurrence of Q in P is within some prefix process. Intuitively, a guarded process cannot affect the behavior of its host process until the action induced by its guarding prefix is performed. New operators are guarded in P if all new processes are guarded in P.

The structural congruence $\equiv_{\mathcal{B}}$ with respect to the buffer store $\mathcal{B}$ is defined as the smallest congruence relation over processes satisfying the following laws:

1. $P \equiv_{\mathcal{B}} Q$, if Q is obtained from P by renaming bound names, or local names not occurring in $\mathcal{B}$.
2. $P \mid Q \equiv_{\mathcal{B}} Q \mid P; P \mid (Q \mid R) \equiv_{\mathcal{B}} (P \mid Q) \mid R; P \mid \mathbf{0} \equiv_{\mathcal{B}} P$.
3. $!P \equiv_{\mathcal{B}} P \mid !P$.
4. $(\nu c)(\nu d)P \equiv_{\mathcal{B}} (\nu d)(\nu c)P$.
5. $(\nu c)\mathbf{0} \equiv_{\mathcal{B}} \mathbf{0}$, if $c \notin ln(\mathcal{B})$; $(\nu c)(P \mid Q) \equiv_{\mathcal{B}} (\nu c)P \mid Q$, if $c \notin ln(\mathcal{B}) \wedge c \notin gn(Q)$.

Structural congruence allows us to pull unguarded new operators to the "outermost" level.

Buffer store $\mathcal{B}$ is *valid* for process P if each local name of $\mathcal{B}$ appears in some new operator occurring at the outermost level of P, i.e., for every $c \in ln(\mathcal{B})$, $P \equiv_{\mathcal{B}} (\nu c)P'$ for some P'.

2.1 Operational Semantics

The (early) transition semantics of π_b is given in terms of a labelled transition system generated by the rules in Table 1. The transition rules are of the form $P, \mathcal{B} \xrightarrow{\alpha} P', \mathcal{B}'$, where P, P' are processes, $\mathcal{B}, \mathcal{B}'$ are buffer stores and α is an action, which can be one of the forms: silent action τ, free input $c(d)$, free output $\overline{c}\langle d\rangle$ or bound output $\overline{c}\langle \nu d\rangle$. We write $n(\alpha)$ for the set of names occurring in α.

Table 1. Transition Rules of π_b

$$\mathtt{IU}\ \frac{c \notin \mathtt{dom}(\mathcal{B})}{c(x).P, \mathcal{B} \xrightarrow{c(d)} P\{d/x\}, \mathcal{B}} \qquad \mathtt{OU}\ \frac{c \notin \mathtt{dom}(\mathcal{B})}{\overline{c}\langle d\rangle.P, \mathcal{B} \xrightarrow{\overline{c}\langle d\rangle} P, \mathcal{B}} \qquad \textsc{Open}\ \frac{P, \mathcal{B}\{c/\nu c\} \xrightarrow{\overline{d}\langle c\rangle} P', \mathcal{B}'}{(\nu c)P, \mathcal{B} \xrightarrow{\overline{d}\langle \nu c\rangle} P', \mathcal{B}'}$$

$$\mathtt{IB}\ \frac{\mathcal{B}(b) = (n, [d] :: l)}{b(x).P, \mathcal{B} \xrightarrow{\tau} P\{d/x\}, \mathcal{B}[b \mapsto (n, l)]} \qquad \mathtt{OB}\ \frac{\mathcal{B}(b) = (n, l);\ |l| < n}{\overline{b}\langle d\rangle.P, \mathcal{B} \xrightarrow{\tau} P, \mathcal{B}[b \mapsto (n, l :: [d])]}$$

$$\mathtt{IBG}\ \frac{\mathcal{B}(b) = (n, l);\ |l| < n;\ b \notin ln(P)}{P, \mathcal{B} \xrightarrow{b(d)} P, \mathcal{B}[b \mapsto (n, l :: [d])]} \qquad \mathtt{OBG}\ \frac{\mathcal{B}(b) = (n, [d] :: l);\ b \notin ln(P)}{P, \mathcal{B} \xrightarrow{\overline{b}\langle d\rangle} P, \mathcal{B}[b \mapsto (n, l)]}$$

$$\textsc{Sum}\ \frac{j \in I;\ \pi_j.P_j, \mathcal{B} \xrightarrow{\alpha} P', \mathcal{B}'}{\sum_{i \in I} \pi_i.P_i, \mathcal{B} \xrightarrow{\alpha} P', \mathcal{B}'} \qquad \textsc{Com}\ \frac{P, \mathcal{B} \xrightarrow{c(d)} P', \mathcal{B};\ Q, \mathcal{B} \xrightarrow{\overline{c}\langle d\rangle} Q', \mathcal{B};\ c \notin \mathtt{dom}(\mathcal{B})}{P \mid Q, \mathcal{B} \xrightarrow{\tau} P' \mid Q', \mathcal{B}}$$

$$\textsc{Par}\ \frac{P, \mathcal{B} \xrightarrow{\alpha} P', \mathcal{B}';\ \text{new operators are guarded in } P \mid Q}{P \mid Q, \mathcal{B} \xrightarrow{\alpha} P' \mid Q, \mathcal{B}'}$$

$$\textsc{New}\ \frac{P, \mathcal{B}\{c/\nu c\} \xrightarrow{\alpha} P', \mathcal{B}';\ c \notin n(\alpha)}{(\nu c)P, \mathcal{B} \xrightarrow{\alpha} (\nu c)P', \mathcal{B}'\{\nu c/c\}} \qquad \textsc{Stru}\ \frac{P \equiv_{\mathcal{B}} P';\ P', \mathcal{B} \xrightarrow{\alpha} Q', \mathcal{B}';\ Q' \equiv_{\mathcal{B}'} Q}{P, \mathcal{B} \xrightarrow{\alpha} Q, \mathcal{B}'}$$

$$\textsc{NewB}\ (\nu b : n)P, \mathcal{B} \xrightarrow{\tau} (\nu b)P, \mathcal{B}[b \mapsto (n, [\,])]$$

These rules are compatible with the transition rules for the π-calculus. IU and OU are rules for unbuffered names and synchronous communication is specified by Com. IB and OB define the asynchronous communication along buffered names: $b(x).P$ performs a τ action by receiving the "oldest" name d from b's buffer, while $\overline{b}\langle d\rangle.P$ performs a τ action by inserting d into b's buffer. Communication along buffered names is asynchronous because it involves two transitions (IB and OB) and other actions may occur between them.

IBG and OBG indicate that a buffer store itself may have actions. If b is a global buffered name, that is (νb) does not occur in P, then we can insert names to or receive names from b's buffer directly. In New and Open, the substitutions on the buffer store are for the sake of validity. NewB is the rule for the extended new process. After creating an empty buffer for b, the capacity parameter n is dropped, leaving the new operator indicating that b is a local name.

The Par rule describes how processes can progress asynchronously, which typically happens with buffered names. However, unlike in the π-calculus, where we have open/close rules to manage name scope extension, in the π_b-calculus, it is hard (perhaps impossible) to define an appropriate close rule because when a local name is exported to a buffer, it becomes hard to track which process will retrieve the name so as to determine the name scope. For instance, consider the process $P_1|P_2|P_3$ where $P_1 = (\nu a)\overline{b}\langle a\rangle.P_1'$, $P_2 = b(y).\cdots$, $P_3 = b(z).\cdots$ and a valid buffer store $\mathcal{B} = [b \mapsto (2, [\,])]$. In the π_b-calculus, P_1 inserts the local a into b's buffer by a τ action, then it can possibly be received by P_2 or P_3, hence tracking the scope of a becomes very hard. Our solution here is to prevent processes from inserting local names into buffers when they are running in parallel with other processes. For processes like the above example, we extend the scope of a to the entire process by structural congruence laws and obtain a process in the form $(\nu a)(\overline{b}\langle a\rangle.P_1'|P_2|P_3)$ thanks to the name convention. This avoids the scope problem.

We have adopted the name convention which simplifies the definition of the labeled transition system. Dealing with names with buffers is subtle and the transition rules without the name convention are presented in [5].

The following proposition says that transition rules preserve buffer validity:

Proposition 1. *If $\mathcal{B}$ is valid for process P and we have the transition $P, \mathcal{B} \xrightarrow{\alpha} P', \mathcal{B}'$, then $\mathcal{B}'$ is valid for P'.*

As in the π-calculus, strong bisimulation over the set of π_b processes can be defined as follows.

Definition 2. *A symmetric binary relation $\mathcal{R}$ over π_b processes is a* bisimulation*, if whenever $(P, \mathcal{B}_P)\mathcal{R}(Q, \mathcal{B}_Q)$ and $(P, \mathcal{B}_P) \xrightarrow{\alpha} (P', \mathcal{B}'_P)$,*

$$\exists(Q', \mathcal{B}'_Q) \,.\, (Q, \mathcal{B}_Q) \xrightarrow{\alpha} (Q', \mathcal{B}'_Q) \wedge (P', \mathcal{B}'_P)\mathcal{R}(Q', \mathcal{B}'_Q)$$

Strong bisimilarity $\dot{\sim}$ is the largest strong bisimulation over the set of π_b processes. $(P, \mathcal{B}_P)$ and $(Q, \mathcal{B}_Q)$ are strongly bisimilar, written as $(P, \mathcal{B}_P) \dot{\sim} (Q, \mathcal{B}_Q)$, if they are related by some strong bisimulation.

2.2 Encoding in the Polyadic π-Calculus

We demonstrate an encoding of the π_b-calculus in the polyadic π-calculus.

Intuitively, a π_b name c is encoded into a pair of π names (c_1, c_2) by the injective *name translation function* N. In the name pair, c_1 is called the *input name* and c_2 the *output name* of c. In addition, input and output names for unbuffered names are identical, but not for buffered names. The two translation names of buffered name b are exactly the names along which a buffer process modelling the buffer of b receives and sends values.

The *translation function* $[\![\cdot]\!]$ takes a π_b process and a valid buffer store as parameters and returns a single π process. The encoding of a buffer store is a composition of buffer processes each representing a buffered name's buffer. For processes, the encoding differs from the original process in the new operators and prefixes. A new operator is encoded into two new operators localizing the pair of translation names. The encoding of input prefix $c(x)$ is also an input prefix but the subject is c's input name c_1, while the encoding of output prefix $\overline{c}\langle d\rangle$ has the output name c_2 as the subject. Finally, in the encoding of an extended new process $(\nu b : n)P$, a buffer process representing b's buffer is added.

The formal definition of the translation, including the buffer process and the translation function, are presented in the companion technical report.

The following lemma shows that transitions of a π_b process can be simulated by its encoding, and no more transition is introduced by the encoding.

Lemma 3. $(P, \mathcal{B}) \xrightarrow{\alpha} (P', \mathcal{B}')$ *if and only if* $[\![P, \mathcal{B}]\!] \xrightarrow{M(\alpha)} [\![P', \mathcal{B}']\!]$.

where M is a bijection relating actions of π_b-calculus to actions of π-calculus. It follows that the encoding preserves strong bisimulation.

Theorem 4. *$(P, \mathcal{B}_P) \dot{\sim} (Q, \mathcal{B}_Q)$ if and only if $[\![P, \mathcal{B}_P]\!] \dot{\sim} [\![Q, \mathcal{B}_Q]\!]$.*

3 The Go Programming Language

The Go programming language is a general purpose language developed by Google to support easy and rapid development of large distributed systems. This section presents a formal operational semantics of the (core) Go language and a fully abstract encoding in the π_b-calculus.

The syntax of a core of Go is presented as follows:

$$\begin{array}{lrl}
\text{Types :} & \mathtt{t} ::= & \mathtt{int} \mid \mathtt{chan}\ \mathtt{t} \\
\text{Expressions :} & e, e_1, e_2, \ldots ::= & x \mid n \mid ch \mid \mathtt{make}(\mathtt{chan}\ \mathtt{t}, n) \mid \texttt{<-}e \\
\text{Statements :} & s, s_1, s_2, \ldots ::= & \mathtt{nil} \mid x = e \mid e_1\texttt{<-}e_2 \mid s_1; s_2 \mid \mathtt{go}\ f(e_1 \ldots e_n) \\
 & & \mid \mathtt{select}\ \{c_1 \ldots c_n\} \\
\text{where} & c_1, c_2, \ldots ::= & \mathtt{case}\ x = \texttt{<-}e : s \mid \mathtt{case}\ e_1\texttt{<-}e_2 : s
\end{array}$$

The *channel type*, coupled with the concept called *Go-routine*, constitutes the core of Go's concurrency system. Channel types are of the form `chan t`, where `t` is called the *element type*. Channels (ch) are first-class values of this language, and they are created by the make expression $\mathtt{make}(\mathtt{chan}\ \mathtt{t}, n)$, where `chan t` specifies the channel type and the integer n specifies the size of the channel buffer. Notice that n must be non-negative and if it is zero, the created channel will be a synchronous channel.

Go-routines are similar to OS threads but much cheaper. A Go-routine is launched by the statement $\mathtt{go}\ f(v_1 \ldots v_n)$. The function body of f will be executed in parallel with the program that executes the `go` statement. When the function completes, this Go-routine terminates and its return value is discarded.

Communication among Go-routines is achieved by sending and receiving operations on channels. Sending statement $ch\texttt{<-}v$ sends v to channel ch, while receiving $\texttt{<-}ch$, regarded as an expression in Go, receives a value from ch. Communication via unbuffered channels are synchronous. Buffered (non-zero sized) channels enable asynchronous communication. Sending a value to a buffered channel can proceed as long as its buffer is not full and receiving from a buffered channel can proceed as long as its buffer is not empty.

`select` statements introduce non-deterministic choice, but their clauses refer to only communication operations. A `select` statement randomly selects a clause whose communication is "ready" (able to proceed), completes the selected communication, then proceeds with the corresponding clause statement.

Without loss of generality, we stipulate that a Go program is a set of function declarations, each of the form $\mathtt{func}\ f(x_1 \ldots x_n)\ \{s\}$. A Go program must specify a main function, which we shall refer to as f_{start} in the sequel, as the entry point — running a Go program is equivalent to executing $\mathtt{go}\ f_{start}(\ldots)$ with appropriate arguments. For the sake of simplicity, we only consider function calls in go statements and we assume that all functions do not return values and their bodies contain no local variables other than function arguments.

3.1 Operational Semantics

The structural operational semantics of Go is defined by a two-level labelled transition system: the local transition system specifies the execution of a single Go-routine in isolation, and the global transition system describes the behavior of a running Go program.

We first define the evaluation of expressions. An expression configuration is a triple $\langle e, \sigma, \delta_c \rangle$, where e is the expression to be evaluated, σ is the *local store* mapping local variables to values, and δ_c is the *channel store* mapping channels to triples (n, l, g), where n is the capacity of the channel's buffer, l is a list of values in the channel buffer, and g is a tag indicating whether the channel is local (0) or global (1). The transition rules between expression configurations $\overset{\alpha}{\longmapsto}_g$ are defined as follows, where actions can be either silent action τ, or $\mathtt{r}(ch, v)$ denoting receive action. We often omit τ from silent transitions.

$$\texttt{VAR}\ \langle x, \sigma, \delta_c \rangle \mapsto_g \langle \sigma(x), \sigma, \delta_c \rangle \qquad \texttt{RvE}\ \frac{\langle e, \sigma, \delta_c \rangle \overset{\alpha}{\longmapsto}_g \langle e', \sigma, \delta'_c \rangle}{\langle \texttt{<-}e, \sigma, \delta_c \rangle \overset{\alpha}{\longmapsto}_g \langle \texttt{<-}e', \sigma, \delta'_c \rangle}$$

$$\texttt{RvU}\ \frac{\delta_c(ch) = (0, [\,], g)}{\langle \texttt{<-}ch, \sigma, \delta_c \rangle \xmapsto{\mathtt{r}(ch,v)}_g \langle v, \sigma, \delta_c \rangle} \quad \texttt{RvB}\ \frac{\delta_c(ch) = (n, [v] :: l, g);\ n > 0}{\langle \texttt{<-}ch, \sigma, \delta_c \rangle \mapsto_g \langle v, \sigma, \delta_c[ch \mapsto (n, l, g)] \rangle}$$

$$\texttt{MAK}\ \frac{ch \notin \mathtt{dom}(\delta_c)}{\langle \mathtt{make(chan\ t}, n), \sigma, \delta_c \rangle \mapsto_g \langle ch, \sigma, \delta_c[ch \mapsto (n, [\,], 0)] \rangle}$$

VAR retrieves the value of x from local store σ. MAK creates a fresh local channel ch. Other rules concern receiving from channels. Once the channel expression is fully evaluated, the real receive begins following rules RvU and RvB. The value received from an unbuffered channel is indicated in the label, while the value received from a buffered channel is the "oldest" value of the channel's buffer.

The local transition system defines transition rules between local configurations. A local configuration is a tuple $\langle s, \sigma, \delta_c \rangle$, where s is the statement to be executed, σ is the *local store* and δ_c is the *channel store*. Each Go-routine has its own local store, but the channel store is shared by all Go-routines of a running program. Some of the local transition rules are presented as follows. Two additional actions can occur in local transition rules: $\mathtt{s}(ch, v)$ for message sending over channels and $\mathtt{g}(f, v_1 \ldots v_n)$ for Go-routine creation.

$$\texttt{SDU}\ \frac{\delta_c(ch) = (0, [\,], g)}{\langle ch\texttt{<-}v, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{s}(ch,v)}_g \langle \mathtt{nil}, \sigma, \delta_c \rangle}$$

$$\texttt{SDB}\ \frac{\delta_c(ch) = (n, l, g);\ n > 0;\ |l| < n}{\langle ch\texttt{<-}v, \sigma, \delta_c \rangle \hookrightarrow_g \langle \mathtt{nil}, \sigma, \delta_c[ch \mapsto (n, l :: [v], g)] \rangle}$$

$$\texttt{GO}\ \langle \mathtt{go}\ f(v_1 \ldots v_n), \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{g}(f, v_1 \ldots v_n)}_g \langle \mathtt{nil}, \sigma, \delta_c \rangle$$

Rules SDU and SDB capture the behavior of sending over unbuffered and buffered channels respectively. Sending a value v over an unbuffered channel ch carries a sending label $\mathtt{s}(ch, v)$, while sending over buffered channels is silent and can proceed as long as the target channel buffer is not full. The Go rule says that a go statement does nothing locally and can always proceed with a transition with the g label — the label is here simply for notifying the global configuration to generate corresponding Go-routines. Subexpression evaluation in Go is strict and leftmost.

Global transitions happen between global configurations which contain information of all running Go-routines. A global configuration, denoted by $\Lambda, \Lambda_1 \ldots$, is defined as a tuple $\langle \Gamma, \delta_c \rangle$, where Γ is a multi-set of statement/local store pairs (s, σ), of all running Go-routines, and δ_c is the channel store. A global transition takes the form $\delta_f \vdash \langle \Gamma_1, \delta_{c_1} \rangle \xrightarrow{\alpha}_g \langle \Gamma_2, \delta_{c_2} \rangle$, where δ_f is a mapping from function names to function definitions. A Go program will start from an initial configuration $\langle \{(s_{start}, \sigma_{start})\}, \delta_{init} \rangle$, where s_{start} is the body of the main function $start$, σ_{start} is the local store of $start$, and δ_{init} is the initial channel store. The global transition rules are listed in Table 2. A global action can be either τ, $\mathtt{r}(ch, v)$ or $\mathtt{s}(ch, v)$.

Table 2. Global Transition Rules

$$\texttt{Loc}\ \frac{\langle s, \sigma, \delta_c \rangle \hookrightarrow_g \langle s', \sigma', \delta_c' \rangle}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \rightarrow_g \langle \Gamma \cup \{(s', \sigma')\}, \delta_c' \rangle}$$

$$\texttt{Com}\ \frac{\langle s_1, \sigma_1, \delta_c \rangle \xhookrightarrow{\mathtt{r}(ch,v)}_g \langle s_1', \sigma_1, \delta_c \rangle;\ \langle s_2, \sigma_2, \delta_c \rangle \xhookrightarrow{\mathtt{s}(ch,v)}_g \langle s_2', \sigma_2, \delta_c \rangle}{\delta_f \vdash \langle \Gamma \cup \{(s_1, \sigma_1), (s_2, \sigma_2)\}, \delta_c \rangle \rightarrow_g \langle \Gamma \cup \{(s_1', \sigma_1), (s_2', \sigma_2)\}, \delta_c \rangle}$$

$$\texttt{LGo}\ \frac{\langle s, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{g}(f, v_1 \ldots v_m)}_g \langle s', \sigma, \delta_c \rangle;\ \delta_f(f) = (\mathtt{func}\ f(x_1 \ldots x_m)\ \{s_f\})}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \rightarrow_g \langle \Gamma \cup \{(s', \sigma), (s_f, [x_1 \mapsto v_1 \ldots x_m \mapsto v_m])\}, \delta_c \rangle}$$

$$\texttt{GRU}\ \frac{\langle s, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{r}(ch,v)}_g \langle s', \sigma, \delta_c \rangle;\ \delta_c(ch) = (0, [\,], 1)}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \xrightarrow{\mathtt{r}(ch,v)}_g \langle \Gamma \cup \{(s', \sigma)\}, \delta_c \rangle}$$

$$\texttt{GRB}\ \frac{\delta_c(ch) = (n, l, 1);\ n > 0;\ |l| < n}{\delta_f \vdash \langle \Gamma, \delta_c \rangle \xrightarrow{\mathtt{r}(ch,v)}_g \langle \Gamma, \delta_c[ch \mapsto (n, l :: [v], 1)] \rangle}$$

$$\texttt{GSU1}\ \frac{\langle s, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{s}(ch,v)}_g \langle s', \sigma, \delta_c \rangle;\ \delta_c(ch) = (0, [\,], 1);\ v \notin \mathtt{dom}(\delta_c)}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \xrightarrow{\mathtt{s}(ch,v)}_g \langle \Gamma \cup \{(s', \sigma)\}, \delta_c \rangle}$$

$$\texttt{GSU2}\ \frac{\langle s, \sigma, \delta_c \rangle \xhookrightarrow{\mathtt{s}(ch,ch')}_g \langle s', \sigma, \delta_c \rangle;\ \delta_c(ch) = (0, [\,], 1);\ \delta_c(ch') = (n', l', g')}{\delta_f \vdash \langle \Gamma \cup \{(s, \sigma)\}, \delta_c \rangle \xrightarrow{\mathtt{s}(ch,\nu ch')}_g \langle \Gamma \cup \{(s', \sigma)\}, \delta_c[ch' \mapsto (n', l', 1)] \rangle}$$

$$\texttt{GSB1}\ \frac{\delta_c(ch) = (n, [v] :: l, 1);\ n > 0;\ v \notin \mathtt{dom}(\delta_c)}{\delta_f \vdash \langle \Gamma, \delta_c \rangle \xrightarrow{\mathtt{s}(ch,v)}_g \langle \Gamma, \delta_c[ch \mapsto (n, l, 1)] \rangle}$$

$$\texttt{GSB2}\ \frac{\delta_c(ch) = (n, [ch'] :: l, 1);\ n > 0;\ \delta_c(ch') = (n', l', g')}{\delta_f \vdash \langle \Gamma, \delta_c \rangle \xrightarrow{\mathtt{s}(ch,\nu v)}_g \langle \Gamma, \delta_c[ch \mapsto (n, l, 1), ch' \mapsto (n', l', 1)] \rangle}$$

`Loc` specifies the independent transition of a single Go-routine. Asynchronous communication will also take this transition since `RvB` and `SdB` are both silent transitions. `LGo` creates a new Go-routine. `Com` defines the synchronous communication between two Go-routines over unbuffered channels. The rules `Loc`, `LGo` and `Com` all specify internal actions of a running program.

A Go program can communicate with the environment via global channels. `GRU`, `GSU1` and `GSU2` describe how a Go program interact with the environment via unbuffered channels, and `GRB`, `GSB1` and `GSB2` describe interactions via buffered channels. Because communication over buffered channels are asynchronous, the labels in `GRB`, `GSB1` and `GSB2` indicate how a global channel interacts with the environment. For instance, in `GRB` the label $\mathtt{r}(ch, v)$ means that the channel (buffer) ch receives a value v from the environment. The two rules `GSU2` and `GSB2` also describe how a local channel is exposed to the environment and becomes a global channel, by communication upon global channels. The ν in the label is required only when the value is a local channel ($g' = 0$).

Let $t = \alpha_1 \ldots \alpha_n$ where each α_i is a global action, we write $\hat{t}$ for the action sequence obtained by eliminating all the occurrences of τ in t. We write $P, \mathcal{B} \xrightarrow{t}_g P', \mathcal{B}'$ if $P, \mathcal{B} \xrightarrow{\alpha_1}_g \cdots \xrightarrow{\alpha_n}_g P', \mathcal{B}'$, and $P, \mathcal{B} \overset{t}{\Rightarrow}_g P', \mathcal{B}'$ if $P, \mathcal{B} \Rightarrow_g \xrightarrow{\alpha_1}_g \Rightarrow_g \cdots \Rightarrow_g \xrightarrow{\alpha_n}_g \Rightarrow_g P', \mathcal{B}'$, where $\Rightarrow_g$ is the reflexive and transitive closure of $\xrightarrow{\tau}_g$.

Definition 5. *A symmetric binary relation $\mathcal{R}$ over global configurations is a* (weak) bisimulation *if $\Lambda_1 \mathcal{R} \Lambda_2$ and $\Lambda_1 \xrightarrow{\alpha}_g \Lambda_1'$ implies $\exists \Lambda_2' \,.\, \Lambda_2 \overset{\hat{\alpha}}{\Rightarrow}_g \Lambda_2' \wedge \Lambda_1' \mathcal{R} \Lambda_2'$. Two global configurations are bisimilar, written as $\Lambda_1 \approx_g \Lambda_2$, if they are related by some bisimulation.*

Two Go programs gp_1, gp_2 are bisimilar, if their initial global configurations (with the same δ_c) are bisimilar.

3.2 Encoding

The encoding of Go in the π_b-calculus is achieved by the translation function $[\![\cdot]\!]_g(r)$, which maps Go expressions and statements to π_b processes. The parameter r is the name along which the result of an expression is returned or the termination of a statement is signaled. Some of the encodings are as follows.

$$\begin{array}{ll}
\textsc{Make} & [\![\mathtt{make(chan}\ \tau, 0)]\!]_g(r) = \underline{\tau}.(\nu a)\overline{r}\langle a\rangle \quad [\![\mathtt{make(chan}\ \tau, n)]\!]_g(r) = \underline{(\nu b : n)}\overline{r}\langle b\rangle \\
\textsc{Recv} & [\![\mathtt{<\text{-}}e]\!]_g(r) = (\nu r')([\![e]\!]_g(r') | r'(y).\underline{y(z)}.\overline{r}\langle z\rangle) \\
\textsc{Send} & [\![e_1\mathtt{<\text{-}}e_2]\!]_g(r) = (\nu r')(LR(e_1, e_2, r') | r'(y, z).\underline{\overline{y}\langle z\rangle}.\overline{r}) \\
\textsc{Go} & [\![\mathtt{go}\ f(e_1 \ldots e_n)]\!]_g(r) = (\nu r')(LR(e_1 \ldots e_n, r') | r'(y_1 \ldots y_n).\underline{\overline{f}\langle y_1 \ldots y_n\rangle}.\overline{r})
\end{array}$$

In the encoding, we use synchronous communication via local names to arrange the evolution order of π_b processes. For instance, in `Recv`, the right hand side of the composition will not proceed unless the left hand side outputs along local name r'.

MAKE returns the local name denoting the newly created channel. A receive operation corresponds to an input prefix in RECV, while a send operation corresponds to an output prefix in SEND. Auxiliary process LR captures the left-to-right evaluation of a sequence of expressions. For the go statement, after evaluating the argument expressions, these arguments are sent to the function to which f refers. The statement does not wait for the function, rather it outputs the termination signal along r immediately.

In the encoding, some prefixes and extended new operators are underlined. They are the most significant part and will be discussed later. The translation function can be extended to a mapping from global configurations (with δ_f) to π_b processes. We write $[\![\Lambda]\!]_g$ for the pair $(P, \mathcal{B})$, where P is the encoding of Λ and δ_f, while $\mathcal{B}$ is a valid buffer store inferred from channel store δ_c.

3.3 Correctness

The correctness of the encoding is demonstrated by a full abstraction theorem with respect to (weak) bisimulation. The following lemma says that a global transition may be simulated by a nontrivial sequence of transitions of its encoding. Usually, the encoding will perform some internal adjustments before and after the real simulation.

Lemma 6. *If $\Lambda \xrightarrow{\alpha}_g \Lambda'$, then $[\![\Lambda]\!]_g \Rightarrow\xrightarrow{M(\alpha)}\Rightarrow [\![\Lambda']\!]_g$, where M is an bijection.*

The lemma is proved by induction on the depth of inference of the premise in the local transition system. Conversely, a sequence of transitions of $[\![\Lambda]\!]_g$ should reflect certain global transitions of Λ. However it is not always possible, since the simulation may not yet complete, even worse the transition sequence simulating one global transition may interleave with transition sequences simulating others. Fortunately, by observing the proof of the previous lemma, we find that actually only one transition in the sequence plays the crucial role, as this transition uniquely identifies a global transition. Other τ transitions, whether preceding or following this special transition, are internal adjustments which prepare for the special transition immediately after them. We call the special transition a *simulating transition*, and the other non-special τ transitions *preparing transitions.*

Definition 7. *A transition $P, B \xrightarrow{\alpha} P', B'$ is a simulating transition if the action α is induced by the underlined prefixes and extended new operators specified in the encoding in Section 3.2. Otherwise, it is a preparing transition.*

Definition 8. *Let Λ be a global configuration, the set $\mathcal{T}_\Lambda$ is defined as follows:*

1. *$[\![\Lambda]\!]_g \in \mathcal{T}_\Lambda$.*
2. *$(P, \mathcal{B}) \in \mathcal{T}_\Lambda$ and $(P, \mathcal{B}) \rightarrow (P', \mathcal{B})$ is a preparing transition, then $(P', \mathcal{B}) \in \mathcal{T}_\Lambda$.*
3. *$(P, \mathcal{B}) \in \mathcal{T}_\Lambda$ and $(P', \mathcal{B}) \rightarrow (P, \mathcal{B})$ is a preparing transition, then $(P', \mathcal{B}) \in \mathcal{T}_\Lambda$.*

Any of the processes in $\mathcal{T}_\Lambda$ can be seen as the encoding of Λ.

Lemma 9. *If $(P, B) \in \mathcal{T}_\Lambda$ and $(Q, B) \in \mathcal{T}_\Lambda$, then we have $(P, B) \approx (Q, B)$.*

As a consequence, bisimulation is preserved by the encoding.

Theorem 10. *$\Lambda_1 \approx_g \Lambda_2$ if and only if $[\![\Lambda_1]\!]_g \approx [\![\Lambda_2]\!]_g$.*

4 Conclusion

We have presented the π_b-calculus which extends the π-calculus by buffered names. Native asynchronous communication is achieved via buffered names. We have provided a fully abstract encoding of an imperative concurrent language in the π_b-calculus with respect to weak bisimulation. A fully abstract translation from Core Erlang to the π_b-calculus can also be obtained. Since Erlang processes are communicated via Erlang mailboxes, the main difference of the encoding of Erlang from that of Go is the explicit modelling of Erlang mailboxes by sequences of buffered names. The details are relegated to the technical report [5].

Acknowledgement. The authors would like to thank Hao Huang for interesting discussion on Erlang.

References

1. Armstrong, J.L.: The Development of Erlang. In: ICFP, pp. 196–203 (1997)
2. Beauxis, R., Palamidessi, C., Valencia, F.D.: On the Asynchronous Nature of the Asynchronous π-Calculus. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 473–492. Springer, Heidelberg (2008)
3. Boudol, G.: Asynchrony and the π-calculus. Rapport de recherche RR-1702, INRIA (1992), `http://hal.inria.fr/inria-00076939`
4. Carlsson, R.: An introduction to Core Erlang. In: PLI 2001 Erlang Workshop (2001)
5. Deng, X., Zhang, Y., Deng, Y., Zhong, F.: The Buffered π-Calculus: A Model for Concurrent Languages (Full version) (2012), `http://arxiv.org/abs/1212.6183`
6. Google Inc.: The Go Programming Language Specification (2012), `http://golang.org/ref/spec`
7. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM 21(8), 666–677 (1978)
8. Honda, K., Tokoro, M.: An Object Calculus for Asynchronous Communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
9. Milner, R.: Communication and Concurrency. PHI Series in computer science. Prentice Hall (1989)
10. Milner, R.: The Polyadic π-Calculus: a tutorial. Tech. Rep. ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh (1991)
11. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Information and Computation 100(1), 1–77 (1992)
12. Noll, T., Roy, C.K.: Modeling Erlang in the π-calculus. In: Erlang Workshop, pp. 72–77 (2005)
13. Roy, C.K., Noll, T., Roy, B., Cordy, J.R.: Towards Automatic Verification of Erlang Programs by π-Calculus Translation. In: Erlang Workshop, pp. 38–50 (2006)
14. Sangiorgi, D., Walker, D.: The π-Calculus: A Theory of Mobile Processes. Cambridge University Press (2001)
15. Walker, D.: Objects in the π-Calculus. Information and Computation 116(2), 253–271 (1995)

Mix-Automatic Sequences

Jörg Endrullis[1], Clemens Grabmayer[2], and Dimitri Hendriks[1]

[1] VU University Amsterdam, The Netherlands
[2] Utrecht University, The Netherlands

Abstract. Mix-automatic sequences form a proper extension of the class of automatic sequences, and arise from a generalization of finite state automata where the input alphabet is state-dependent. In this paper we compare the class of mix-automatic sequences with the class of morphic sequences. For every polynomial φ we construct a mix-automatic sequence whose subword complexity exceeds φ. This stands in contrast to automatic and morphic sequences which are known to have at most quadratic subword complexity. We then adapt the notion of k-kernels to obtain a characterization of mix-automatic sequences, and employ this notion to construct morphic sequences that are not mix-automatic.

1 Introduction

Automatic sequences [1] were introduced by Cobham [4] in 1972, and have since been been studied extensively. A sequence $w : \mathbb{N} \to \Delta$ over a finite alphabet Δ is *automatic* if it can be realized by a finite automaton that, for some $k \geq 2$, takes the base-k expansion $(n)_k$ of a number $n \in \mathbb{N}$ as input and outputs the n-th letter of w; in this case w is called *k-automatic*. For multiplicatively independent k and ℓ, k-automaticity and ℓ-automaticity are almost separated notions; e.g., if a sequence is both 2-automatic and 3-automatic, then it is ultimately periodic.

Therefore it is natural to study also nonstandard numeration systems, and the classes of automatic sequences they give rise to. Rigo [10] and Rigo and Maes [11] study 'abstract numeration systems' based on the 'shortlex' order on an infinite regular language, induced by an order on the alphabet. With this concept they precisely capture the class of morphic sequences.

We introduce *dynamic radix* numeration systems which are obtained as a natural generalization from another variation of the standard base-k representation: the *mixed radix* numeration systems [8] in which the base used only depends on the position of a digit. In dynamic radix numeration systems the base used may depend on the input digits read so far. Sequences realized by finite automata that take dynamic radix input we call *mix-automatic*.

We first consider an example of a 2-automatic sequence, the celebrated Thue–Morse sequence, and explain how it is generated by the automaton in Figure 1. The automaton has states $\{q_0, q_1\}$, initial state q_0, input alphabet $\{0, 1\}$ and output alphabet $\{a, b\}$. The output letter assigned to q_0 is a and to q_1 is b (indicated by *state/output* in the states of the automaton). The n-th letter of the sequence is the output of the automaton when reading $(n)_2$, the base-2

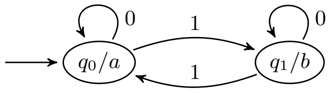

$$\begin{tikzcd}
\arrow[r] & q_0/a \arrow[loop above, "0"] \arrow[r, bend left, "1"] & q_1/b \arrow[loop above, "0"] \arrow[l, bend left, "1"]
\end{tikzcd}$$

Fig. 1. DFAO generating the Thue–Morse sequence $abbabaabbaababba\cdots$

expansion of n. For example, for input $(3)_2 = 11$ the automaton ends in state q_0 with output a, and for input $(4)_2 = 100$ in state q_1 with output b.

The automaton of Figure 1 is called a *deterministic finite-state automaton with output (DFAO)*. For $k \geq 2$, a k-DFAO is an automaton over the input alphabet $\mathbb{N}_{<k} = \{0, 1, \ldots, k-1\}$. An infinite sequence $w \in \Delta^\omega$ is called *k-automatic* if there exists a k-DFAO such that for every $n \in \mathbb{N}$ the output of the automaton when reading the word $(n)_k \in \mathbb{N}^*_{<k}$ is $w(n)$, with $(n)_k$ the base-k expansion of n.

Mix-Automatic Sequences. The class of automatic sequences is well-known to have good closure properties; for example, it is closed under shifts (prepending letters or removing prefixes), and taking arithmetic subsequences. The class of mix-automatic sequences extends the class of automatic sequences, has all these closure properties, and additionally is closed under k-shuffling, for all $k \geq 2$.

Mix-automatic sequences are defined via *mix-DFAOs*, automata that generalize k-DFAOs by allowing that the alphabet of the symbol to be processed next depends on the current state. Let us consider the example of a mix-DFAO shown in Figure 2. The state q_0 has two outgoing edges, reflecting the input alphabet $\{0, 1\}$, while q_1 has three outgoing edges, reflecting the input alphabet $\{0, 1, 2\}$.

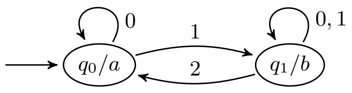

$$\begin{tikzcd}
\arrow[r] & q_0/a \arrow[loop above, "0"] \arrow[r, bend left, "1"] & q_1/b \arrow[loop above, "{0,1}"] \arrow[l, bend left, "2"]
\end{tikzcd}$$

Fig. 2. An example of a mix-DFAO

Dynamic Radix Numeration Systems. Clearly, the numeration system used for the input of mix-DFAOs cannot be the standard base-k representation. Instead, in the number representation that we let these automata operate on, the base for each digit is determined by the lower-significance digits that have already been read.Thus we let the automata read from the least to the most significant digit (i.e., we let the reading direction be from right to left). We write $(n)_M$ for the number representation of n that serves as input for the automaton M. For M the automaton from Figure 2, the representations of the first eight numbers are

$$\begin{array}{llll}
(0)_M = \varepsilon & (2)_M = 1_2 0_2 & (4)_M = 1_2 0_2 0_2 & (6)_M = 1_3 1_2 0_2 \\
(1)_M = 1_2 & (3)_M = 1_3 1_2 & (5)_M = 2_3 1_2 & (7)_M = 1_3 0_3 1_2
\end{array}$$

where a subscript b (not part of the number representation) in d_b indicates the base employed for d. Let us explain this at the example $(17)_M = 1_2 0_2 2_3 1_2$.

Knowing the base for each digit, we can reconstruct the value of the representation as follows: $17 = 1 \cdot 2 \cdot 3 \cdot 2 + 0 \cdot 3 \cdot 2 + 2 \cdot 2 + 1$ where each digit is multiplied with the product of the bases of the lower digits. Given just the representation 1021, the base of each of the digits is determined by the input alphabet of the state of the automaton reading the digit. The states q_0 and q_1 of M have input alphabets $\{0, 1\}$ and $\{0, 1, 2\}$ and thus expect the input in base 2 and 3, respectively. When reading 1021 (right to left) the automaton M visits the states q_0, q_1, q_0, q_0 and q_1. Annotating the input digits with the state of the automaton when reading the digit, we obtain $1_{q_0} 0_{q_0} 2_{q_1} 1_{q_0}$, and taking into account the bases expected by these states, yields $1_2 0_2 2_3 1_2$.

We emphasize that, given a mix-DFAO M, every $n \in \mathbb{N}$ has a unique representation $(n)_M = d_m \cdots d_0$ (without leading zeros). This representation can be computed as follows. Assume that we have determined the value of the digits $d_{i-1} \cdots d_0$ with corresponding bases $b_{i-1} \cdots b_0$. The base b_i of digit d_i is determined by the input alphabet of the state of the automaton after reading $d_{i-1} \cdots d_0$ (right to left), and digit d_i is the remainder of the division of $n - \sum_{0 \le j < i} d_j (b_{j-1} \cdots b_1 \cdot b_0)$ by b_i.

Every mix-DFAO M gives rise to a mix-automatic sequence $w \in \Delta^\omega$ by defining for every $n \in \mathbb{N}$, $w(n)$ as the output of M when reading $(n)_M$.

Zip-Specifications. In [6] it has been shown that k-automatic sequences are precisely the class of sequences definable by *zip-k specifications*, that is, systems of recursion equations $\{X_1 = t_1, \ldots, X_n = t_n\}$ with terms t_i built from the syntax

$$t ::= X_i \mid a : t \mid \mathsf{zip}_k(t, \ldots, t) \qquad\qquad (1 \le i \le n,\ a \in \Delta)$$

Semantically, the term notation $a : t$ indicates the concatenation of a letter with a sequence, and the k-ary symbol zip_k stands for the function of type $(\Sigma^\omega)^k \to \Sigma^\omega$ that zips (or interleaves or shuffles) its k argument sequences, and is defined by

$$\mathsf{zip}_k(w_0, \ldots, w_{k-1})(kn + i) = w_i(n) \qquad\qquad (0 \le i < k)$$

Operationally, zip_k can be defined by the rewrite rule

$$\mathsf{zip}_k(x : t_0, t_1, \ldots, t_{k-1}) \to x : \mathsf{zip}_k(t_1, \ldots, t_{k-1}, t_0) \qquad (1)$$

The zip operation on finite words is known in the literature as *perfect shuffle* [2]. An example of a zip-2 specification corresponding to the 2-DFAO from Fig. 1 is

$$\mathsf{M} = a : \mathsf{Q}_1 \qquad \mathsf{Q}_0 = a : \mathsf{zip}_2(\mathsf{Q}_0, \mathsf{Q}_1) \qquad \mathsf{Q}_1 = b : \mathsf{zip}_2(\mathsf{Q}_1, \mathsf{Q}_0) \qquad (2)$$

The Thue–Morse sequence is the unique solution for the variable M in this specification, or, from a rewriting perspective, it is the infinite normal form of M in the rewrite system consisting of (1) and (2), orienting the equations from left to right. For further details we refer to [6].

The introduction of mix-automatic sequences was motivated by the characterization of k-automatic sequences as the class of sequences that can defined by zip-k specifications, answering the question: *What class of sequences is obtained when allowing zips of different arities in the same specification?* In [6,7] the correspondence between such 'zip-mix' specifiable sequences and mix-automatic sequences was established. Moreover, it was shown that mix-automaticity properly extends automaticity: for example, shuffling a 2-automatic and a 3-automatic sequence, both not ultimately periodic, is mix-automatic but not automatic.

Contribution and Overview. We continue the study of mix-automatic sequences started in [6,7] by exploring the relationship with morphic sequences. In Section 3, we generalize the characterization of k-automatic sequences via finite k-kernels to the setting of mix-automatic sequences. In Sections 4 and 5 we show that neither of the classes (a) mix-automatic sequences and (b) morphic sequences subsumes the other. In particular we show that the subword complexity of mix-automatic sequences can exceed any polynomial, whereas it is known [5] that morphic sequences have at most quadratic subword complexity.

2 Preliminaries

We use standard terminology and notation; for example, see Allouche and Shallit [1]. Let Σ be a finite alphabet. Then we denote by

- Σ^* the set of all *finite words over* Σ, by ε the *empty word*,
- $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ the set of *finite non-empty words*,
- $\Sigma^\omega = \{w \mid w : \mathbb{N} \to \Sigma\}$ the set of *infinite words over* Σ,
- $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ the set of all *(finite or infinite) words.*

For a word $w \in \Sigma^\infty$ and $n \in \mathbb{N}$, we write $w(n)$ for the n-th letter of w (counting from zero). We write $|x|$ for the *length of* $x \in \Sigma^\infty$, with $|x| = \infty$ if x is infinite. We call a word $v \in \Sigma^*$ a *subword* of $x \in \Sigma^\infty$ if $x = uvy$ for some $u \in \Sigma^*$ and $y \in \Sigma^\infty$, and say that v *occurs at position* $|u|$. The *subword complexity* of a sequence $w \in \Sigma^\omega$ is the function $p_w : \mathbb{N} \to \mathbb{N}$ such that $p_w(n)$ is the number of distinct length-n subwords (factors) of w.

Definition 1. A *deterministic finite automaton with output (DFAO)* is a tuple $\langle Q, \Sigma, \delta, q_0, \Delta, \lambda \rangle$ where

- Q is a finite set of states with $q_0 \in Q$ the initial state,
- Σ a finite input alphabet, Δ an output alphabet,
- $\delta : Q \times \Sigma \to Q$ a transition function, and
- $\lambda : Q \to \Delta$ an output function.

We extend the domain of δ to $Q \times \Sigma^*$ by defining, for all $q \in Q$, $\delta(q, \varepsilon) = q$ and

$$\delta(q, xa) = \delta(\delta(q, a), x) \qquad \text{for all } x \in \Sigma^* \text{ and } a \in \Sigma,$$

thus forcing the reading direction from right to left.

For $n, k \in \mathbb{N}$, $k \geq 2$, we let $(n)_k$ denote the canonical *base-k expansion of n* (without leading zeros). More precisely, for $n > 0$ we have

$$(n)_k = d_m d_{m-1} \cdots d_0 \quad \text{where} \quad 0 \leq d_0, \dots, d_m < k,\ d_m > 0 \text{ and } n = \sum_{i=0}^{m} d_i k^i \,.$$

For $n = 0$ we fix $(n)_k = \varepsilon$. We emphasize that the exclusion of leading zeros in the number representation $(n)_k$ is not crucial. Every DFAO can be transformed into an equivalent DFAO that ignores leading zeros, see [1].

Definition 2. Let $k \geq 2$ and define $\mathbb{N}_{<k} = \{0, \dots, k-1\}$. A k-DFAO M is a DFAO $\langle Q, \Sigma, \delta, q_0, \Delta, \lambda \rangle$ with the input alphabet $\Sigma = \mathbb{N}_{<k}$.

For $q \in Q$, we define the infinite sequence $\mathrm{seq}(M, q) \in \Delta^\omega$ by $\mathrm{seq}(M, q)(n) = \lambda(\delta(q, (n)_k))$, for every $n \in \mathbb{N}$. We write $\mathrm{seq}(M)$ as shorthand for $\mathrm{seq}(M, q_0)$. The automaton M is said to *generate* the sequence $\mathrm{seq}(M)$.

Now automatic sequences can be defined as follows:

Definition 3. A sequence $w \in \Delta^\omega$ is *k-automatic* if there exists a k-DFAO that generates w. A sequence is called *automatic* if it is k-automatic for some $k \geq 2$.

3 Mix-Automatic Sequences

In this section we introduce mix-automatic sequences. For this purpose, we define finite automata (with output) that have state-dependent input alphabets. As inputs these automata take dynamic radix number representations, which generalize base-k number representations to the effect that the digits are allowed to belong to different bases, and may depend on previously read digits. For specifying the format of the dynamic radix number representation that an automaton can process we use 'base determiners', which are themselves finite automata with (number) output that determine the base of each digit depending on the values of the lower digits. Number representations according to a thus obtained dynamic radix number representation can then serve as inputs for a mix-DFAO. k-DFAOs are special cases of mix-DFAOs. Eventually, we introduce mix-automatic sequences as sequences that are generated by mix-DFAOs.

Deterministic Finite State Automata with State-Dependent Input Alphabet. We introduce finite automata with output for which the input alphabet is dependent on the current state.

Definition 4. A *state-dependent input alphabet DFAO* is a tuple of the form $\langle Q, \Sigma, \delta, q_0, \Delta, \lambda \rangle$ where

- Q is a finite set of states with $q_0 \in Q$ the initial state,
- $\Sigma = \{\Sigma_q\}_{q \in Q}$ is a family of input alphabets,
- $\delta = \{\delta_q : \Sigma_q \to Q\}_{q \in Q}$ is a family of transition functions,

- Δ is an output alphabet, and
- $\lambda : Q \to \Delta$ is an output function.

We interpret δ as a partial function $Q \times \bigcup \Sigma \rightharpoonup Q$, and define $\delta(q,i) = \delta_q(i)$ iff $i \in \Sigma_q$. We extend the domain of δ to $Q \times (\bigcup \Sigma)^*$ by defining for all $q \in Q$, $\delta(q,\varepsilon) = q$, and for all $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma_q$

$$\delta(q, xa) = \delta(\delta(q,a), x) \qquad\qquad \text{if } \delta(\delta(q,a),x) \text{ is defined.}$$

Note that the definition of δ forces the reading direction of input words to be from right to left. An alternative definition of δ is as follows: Let $q \in Q$ and $w = a_{n-1} \cdots a_0$ where $a_i \in \Sigma_{r_i}$ $(0 \leq i < n)$ with $r_i \in Q$ defined (for $0 \leq i \leq n$) by $r_0 = q$ and $r_{i+1} = \delta(r_i, a_i)$; then we set $\delta(q,w) = r_n$.

The following definition generalizes k-DFAOs:

Definition 5. A *mix-DFAO* is a tuple $\langle Q, \beta, \delta, q_0, \Delta, \lambda\rangle$ that represents a state-dependent input alphabet DFAO $\langle Q, \{\mathbb{N}_{<\beta(q)}\}_{q\in Q}, \delta, q_0, \Delta, \lambda\rangle$ with $\beta : Q \to \mathbb{N}_{\geq 2}$.

Obviously, mix-DFAOs require a special number representation as input. The number representation must ensure that the base of each digit matches the input alphabet of the state the automaton is in when reading the digit. This leads to the following generalization of the usual base-k number representations.

Dynamic Radix Numeration Systems and Base Determiners. We now introduce dynamic radix number representations. For defining these representations special mix-DFAOs called 'base determiners' are used to specify the base for each digit depending on the digits that have been read before.

Definition 6. A *base determiner* is a tuple $\langle Q, \beta, \delta, q_0\rangle$ which is a shorthand for the mix-DFAO $\langle Q, \beta, \delta, q_0, \mathbb{N}, \beta\rangle$. The base determiner *underlying* a mix-DFAO $\langle Q, \beta, \delta, q_0, \Delta, \lambda\rangle$ is the base determiner $\langle Q, \beta, \delta, q_0\rangle$.

Let $B = \langle Q, \beta, \delta, q_0\rangle$ be a base determiner. The *base-B representation* of an integer $n \in \mathbb{N}$ is defined by $(n)_B = (n)_{q_0}$ where $(0)_q = \varepsilon$ and for $n > 0$

$$(n)_q = (n')_{\delta(q,d)}\, d\,, \qquad n' = \lfloor n/\beta(q) \rfloor\,, \quad \text{and} \quad d = n - n' \cdot \beta(q)$$

So n' and d are quotient and remainder of division of n by $\beta(q)$, respectively.

Definition 7. Let $B = \langle Q, \beta, \delta, q_0\rangle$ be a base determiner. We define the partial function $[\_]_B : \mathbb{N}^* \rightharpoonup \mathbb{N}$ by $[w]_B = [w, 1]_{q_0}$ where we let $[w, b]_q$ for all $b \in \mathbb{N}$ and $q \in Q$ be defined by

$$[\varepsilon, b]_q = 0 \qquad\qquad [wd, b]_q = [w, b\beta(q)]_{\delta(q,d)} + bd \qquad \text{if } d \in \mathbb{N}_{<\beta(q)}$$

and undefined otherwise.

Note that $[\_]_B$ is the left inverse of $(\_)_B$: for all $b \in \mathbb{N}$ and $q \in Q$ $[(n)_q, b]_q = bn$ follows by induction on $n \in \mathbb{N}$.

We obtain ordinary base-k numbers by defining the base determiner B to consist of a single state q with output k and edges $0, \ldots, k-1$ looping to itself; this is illustrated in Figure 3.

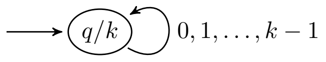
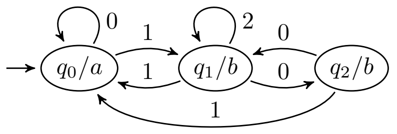


Fig. 3. A base determiner for the standard base-k number representation

Example 8. Consider the following mix-DFAO M and the dynamic numeration system it defines (where $n > 0$, and $q \in \{q_0, q_1, q_2\}$):



$$\begin{aligned} (2n)_{q_0} &= (n)_{q_0}0 & (0)_q &= \varepsilon \\ (2n+1)_{q_0} &= (n)_{q_1}1 & (3n)_{q_1} &= (n)_{q_2}0 \\ (2n)_{q_2} &= (n)_{q_1}0 & (3n+1)_{q_1} &= (n)_{q_0}1 \\ (2n+1)_{q_2} &= (n)_{q_0}1 & (3n+2)_{q_1} &= (n)_{q_1}2 \end{aligned}$$

Let B be the base determiner underlying M (that is, obtained from M by redefining the output for q_0, q_1 and q_2 as 2, 3 and 2, respectively).

As an example, we compute $(5)_B$, and $(23)_B$ as follows:

$$\begin{aligned} (5)_B &= (5)_{q_0} = (2)_{q_1}1 = (0)_{q_2}21 = 21 \\ (23)_B &= (23)_{q_0} = (11)_{q_1}1 = (3)_{q_1}21 = (1)_{q_2}021 = (0)_{q_0}1021 = 1021\ . \end{aligned}$$

A k-DFAO is an automaton reading the input in the base-k number format. We generalize this concept to B-DFAOs that expect to read input in the number format defined by the base determiner B.

Definition 9. Let M be a mix-DFAO and B a base determiner. We call M a *B-DFAO* if M is *compatible with* B in the sense that $(n)_B = (n)_{B_M}$ holds for all $n \in \mathbb{N}$, where B_M is the base determiner underlying M.

(Note that M is a B_M-DFAO, i.e., M reads the number format defined by itself.)

A B-DFAO with output alphabet Δ defines a B-automatic sequence $w \in \Delta^\omega$ by defining for all $n \in \mathbb{N}$, $w(n)$ as the output of the DFAO on the input $(n)_B$. Sequences generated by mix-DFAOs we call 'mix-automatic' sequences.

Definition 10. Let B be a base determiner, and $M = \langle Q, \beta, \delta, q_0, \Delta, \lambda \rangle$ a B-DFAO. For states $q \in Q$, we define $\mathrm{seq}(M, q) \in \Delta^\omega$ by:

$$\mathrm{seq}(M, q)(n) = \lambda(\delta(q, (n)_B)) \qquad \text{for all } n \in \mathbb{N}$$

We define $\mathrm{seq}(M) = \mathrm{seq}(M, q_0)$, and say M *generates* the sequence $\mathrm{seq}(M)$.

A sequence $w \in \Delta^\omega$ is *B-automatic* if there exists a B-DFAO M such that $w = \mathrm{seq}(M)$. A sequence is called *mix-automatic* if it is B-automatic for some base determiner B.

Example 11. We continue Example 8. The sequence $\mathrm{seq}(M)$ begins with

$$abbab\underline{b}aabbbaaaabbbbbbaa\underline{a}abababbbbbbabababaaaaaababbbabbbabbb\cdots$$

with entries 5 and 23 underlined. E.g. $\lambda(\delta(q_0, 1021)) = a$ since starting from q_0 and reading 1021 from right to left brings you back at state q_0 with output a.

Kernels for Mix-Automatic Sequences. Automatic sequences can be characterized in terms of their 'kernels' being finite. For sequences $w \in \Delta^\omega$ and $i, k \in \mathbb{N}$, $k > 0$ we define

$$\pi_{i,k}(w) = w(i)\, w(i+k)\, w(i+2k)\, w(i+3k)\, w(i+4k) \cdots ,$$

the subsequence of w selecting every k-th element starting form the i-th element (counting from 0). The *k-kernel* $\mathrm{Ker}(k, w)$ of a sequence $w \in \Delta^\omega$ is the set of arithmetic subsequences $\mathrm{Ker}(k, w) = \{\pi_{i,k^p}(w) \mid p \in \mathbb{N}, i < k^p\}$. The set $\mathrm{Ker}(k, w)$ can equivalently be defined as the smallest set K such that $w \in K$, and for all $u \in K$ we have $\pi_{i,k}(u) \in K$ for all $0 \le i < k$; see further [6].

Fact ([1, Thm. 6.6.2]). *A sequence is k-automatic iff its k-kernel is finite.*

We now generalize this characterization to mix-automatic sequences.

Definition 12. Let $x : \Delta^\omega \to \mathbb{N}_{\ge 2}$. The *$x$-kernel* $\mathrm{Ker}(x, w)$ of a sequence $w \in \Delta^\omega$ is defined as the smallest set $K \subseteq \Delta^\omega$ such that $w \in K$, and for all sequences $u \in K$ we have $\pi_{i,x(u)}(u) \in K$ for all $0 \le i < x(u)$.

The function $x : \Delta^\omega \to \mathbb{N}_{\ge 2}$ determines for every sequence $w \in \Delta^\omega$ the set of derivative functions $\{\pi_{0,x(w)}, \pi_{1,x(w)}, \ldots, \pi_{x(w)-1,x(w)}\}$ to be applied to w. The ordinary k-kernels ($k \in \mathbb{N}$) are obtained by defining $x(w) = k$ for every $w \in \Delta^\omega$.

Theorem 13. *A sequence $w \in \Delta^\omega$ is mix-automatic if and only if there exists a function $x : \Delta^\omega \to \mathbb{N}_{\ge 2}$ such that the x-kernel of w is finite.*

Proof. We show the less obvious direction, from left to right. For this let $M = \langle Q, \beta, \delta, q_0, \Delta, \lambda \rangle$ be a mix-DFAO that generates a sequence w. For every state $q \in Q$ the equality $\mathrm{seq}(M, q) = \mathsf{zip}_{\beta(q)}(\mathrm{seq}(M, \delta(q, 0)), \ldots, \mathrm{seq}(M, \delta(q, \beta(q) - 1)))$ holds, that is, the sequence generated by a state q is the shuffling of the sequences generated by the successor states of q. As a consequence, whenever M contains states $q_1 \neq q_2 \in Q$, $q_2 \neq q_0$ with $\mathrm{seq}(M, q_1) = \mathrm{seq}(M, q_2)$ we can eliminate q_2 after redirecting all its incoming edges to q_1; this changes the number representation, but leaves the sequence generated by the automaton unaltered. Thus we may assume that $\mathrm{seq}(M, q_1) \neq \mathrm{seq}(M, q_2)$ for all $q_1 \neq q_2 \in Q$. Hence we can define the function $x : \Delta^\omega \to \mathbb{N}_{\ge 2}$ as follows: $x(\mathrm{seq}(M, q)) = \beta(q)$ for every $q \in Q$, and $x(u) = 2$ for all other sequences u. Then it follows immediately that $\mathrm{Ker}(x, w) \subseteq \{\mathrm{seq}(M, q) \mid q \in Q\}$, namely, the set of sequences generated by the reachable states, and that $\mathrm{Ker}(x, w)$ is finite. $\square$

We refine this characterization with respect to a given number representation.

Definition 14. Let $B = \langle Q, \beta, \delta, q_0 \rangle$ be a base determiner. The *B-kernel* of a sequence $w \in \Delta^\omega$, which is denoted by $\mathrm{Ker}(B, w)$, is the set $\{u \mid (u, q) \in K\}$ where $K \subseteq \Delta^\omega \times Q$ is the smallest set such that $(w, q_0) \in K$, and $(\pi_{i,\beta(q)}(u), \delta(q, i)) \in K$ for all $(u, q) \in K$ and $i \in \mathbb{N}$ with $0 \le i < \beta(q)$.

Theorem 15. *A sequence $w \in \Delta^\omega$ is B-automatic iff its B-kernel is finite.*

4 The Subword Complexity of Mix-Automatic Sequences

We show for any polynomial φ there exists a mix-automatic sequence with a subword complexity exceeding φ. It immediately follows that there are mix-automatic sequences that are not morphic. This answers a question of [6].

For $p, n \in \mathbb{N}_{>0}$ with p a prime number, we use $\nu_p(n)$ to denote the *p-adic valuation of n*, that is, the largest integer $k \in \mathbb{N}$ such that p^k divides n. For every prime number p, we define the sequence $\gamma_p \in \{0,1\}^\omega$ by

$$\gamma_p = (\nu_p(1) \bmod 2)\,(\nu_p(2) \bmod 2)\,(\nu_p(3) \bmod 2)\,\cdots$$

The sequence γ_2 is the well-known *period-doubling sequence* [1, Example 6.4.3]:

$$\gamma_2 = 0100010101000100010001010100010101000101010001000100010101 00\cdots$$

We show that shuffling k sequences from the set $\{\gamma_p \mid p \text{ is prime}\}$ yields a mix-automatic sequence with subword complexity in $\Omega(n^k)$. We first show that each of the sequences has at least linear subword complexity:

Lemma 16. *The subword complexity of γ_p is in $\Omega(n)$ for every prime number p.*

Proof. The Morse–Hedlund theorem [9] asserts that an infinite sequence w is ultimately periodic if and only if for some $n \in \mathbb{N}$ not more than n factors of length n occur in w. Hence, it suffices to show that γ_p is not ultimately periodic. Assume that γ_p would be ultimately periodic. Then there exist $n_0, k > 1$ such that $\nu_p(n) \equiv \nu_p(n+k) \pmod 2$ for every $n \geq n_0$. Let $n = p^{\nu_p(k)+2m+1}$ with $m \in \mathbb{N}$ such that $n \geq n_0$. Then $\nu_p(n) = \nu_p(k) + 2m + 1$ and $\nu_p(n+k) = \nu_p(k)$, and hence $\nu_p(n) \not\equiv \nu_p(n+k) \pmod 2$ contradicting the assumption. $\square$

We moreover employ that the sequences have the following regular structure:

Lemma 17. *Let p be a prime number, $k \in \mathbb{N}$ and w the prefix of length $p^k - 1$ of the sequence γ_p. Then w occurs in γ_p at every position $n \cdot p^k$ ($n \in \mathbb{N}$).*

Proof. Let $0 \leq i < p^k - 1$. Then we have $\gamma_p(n \cdot p^k + i) \equiv \nu_p(n \cdot p^k + i + 1) \pmod 2$ and $\nu_p(n \cdot p^k + i + 1) = \nu_p(i+1)$ for every $n \in \mathbb{N}$. $\square$

Lemma 18. *Let $k > 0$, $p_1, \ldots, p_k$ be pairwise distinct primes. Then the sequence $\mathsf{zip}_k(\gamma_{p_1}, \gamma_{p_2}, \ldots, \gamma_{p_k})$ is mix-automatic and its subword complexity is in $\Omega(n^k)$.*

Proof. By Lemma 16 the sequences $\gamma_{p_1}, \ldots, \gamma_{p_k}$ have subword complexity in $\Omega(n)$. Hence, for proving that the subword complexity of $\mathsf{zip}_k(\gamma_{p_1}, \gamma_{p_2}, \ldots, \gamma_{p_k})$ is in $\Omega(n^k)$, it suffices to show the following: for every $n \in \mathbb{N}$ whenever $w_1, \ldots, w_k$ are n-length subwords of the sequences $\gamma_{p_1}, \ldots, \gamma_{p_k}$, respectively, the shuffle $\mathsf{zip}_k(w_1, \ldots, w_k)$ of length kn is a subword of $\mathsf{zip}_k(\gamma_{p_1}, \gamma_{p_2}, \ldots, \gamma_{p_k})$.

For this purpose, we show ($*$) there exists a position $q \in \mathbb{N}$ such that for all i ($1 \leq i \leq k$), the word w_i occurs in γ_{p_i} at position q. Let $\ell_1, \ldots, \ell_k$ be such that every w_i ($1 \leq i \leq k$) occurs in the prefix of γ_{p_i} of length $p_i^{\ell_i} - 1$. Let $o_1, \ldots, o_k$ be the positions of the first occurrences of $w_1, \ldots, w_k$ in $\gamma_{p_1}, \ldots, \gamma_{p_k}$, respectively. (All of these positions are in the respective prefixes of γ_{p_i} of length $p_i^{\ell_i} - 1$.) We proceed by induction on $1 \leq i \leq k$ to construct integers $a_i, b_i > 0$ such that

(i) for all $1 \leq j \leq i$, the word w_j occurs at all positions $a_i + m \cdot b_i$ $(m \in \mathbb{N})$, and
(ii) for all $i < j \leq k$, b_i is coprime with p_j, i.e., $\gcd(b_i, p_j) = 1$.

For $i = 1$, we choose $a_1 = o_1$ and $b_1 = p_1^{\ell_1}$. Then, as a consequence of Lemma 17, the word w_1 occurs at every position $a_1 + m \cdot b_1$ $(m \in \mathbb{N})$.

Let $i < k$ and $c_{i+1} = p_{i+1}^{\ell_{i+1}}$. From (ii) it follows that b_i and c_{i+1} are coprime. By Euler's theorem, there exists $1 \leq e_{i+1} \in \mathbb{N}$ such that $b_i^{e_{i+1}} \equiv 1 \pmod{c_{i+1}}$. As a consequence we can find some $0 \leq a_i' < c_{i+1}$ and define $a_{i+1} = a_i + a_i' \cdot b_i^{e_{i+1}}$ such that $a_{i+1} \equiv o_{i+1} \pmod{c_{i+1}}$. We let $b_{i+1} = c_{i+1} \cdot b_i^{e_{i+1}}$. Then we have:

$$a_{i+1} = a_i + (a_i' \cdot b_i^{e_{i+1}-1}) \cdot b_i \qquad\qquad b_{i+1} = (c_{i+1} \cdot b_i^{e_{i+1}-1}) \cdot b_i$$

We have $\{a_{i+1} + m \cdot b_{i+1} \mid m \in \mathbb{N}\} \subseteq \{a_i + m \cdot b_i \mid m \in \mathbb{N}\}$, and hence for every $1 \leq j \leq i + 1$, the word w_j occurs in γ_{p_j} at all positions $a_{i+1} + m \cdot b_{i+1}$ with $m \in \mathbb{N}$. Moreover $a_{i+1} + m \cdot b_{i+1} \equiv o_{i+1} \pmod{c_{i+1}}$ for every $m \in \mathbb{N}$, and thus by Lemma 17, w_{i+1} occurs in $\gamma_{p_{i+1}}$ at all positions $a_{i+1} + m \cdot b_{i+1}$ with $m \in \mathbb{N}$. We have that $b_{i+1} = (c_{i+1} \cdot b_i^{e_{i+1}}) = p_{i+1}^{\ell_{i+1}} \cdot b_i^{e_{i+1}}$ and thus b_{i+1} and b_j are coprime for every $i + 1 < j \leq k$.

Finally, we define $q = a_k$ and by induction hypothesis (i) we have $(*)$. □

Morphic sequences have at most quadratic subword complexity [5]. Hence, by Lemma 18 the mix-automatic sequences $\mathsf{zip}_k(\gamma_{p_1}, \gamma_{p_2}, \ldots, \gamma_{p_k})$ for $k > 2$ are not morphic.

Theorem 19. *The class of mix-automatic sequences is not contained in the class of morphic sequences.*

5 Morphic Sequences That Are Not Mix-Automatic

In the previous section, we have seen that the class of mix-automatic sequences is not contained in the class of morphic sequences. We now show that the reverse holds as well, that is, there exist morphic sequences that are not mix-automatic. In particular, we consider the characteristic sequence of (positive) squares:

$$\begin{aligned} \mathsf{squares} &= 100100001000000100000000100000000001000000000001\cdots \\ &= 10^2 10^4 10^6 10^8 10^{10} 10^{12} 10^{14} 1 \cdots \end{aligned}$$

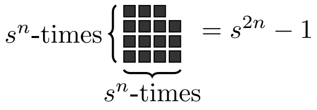
So $\mathsf{squares} \in \{0,1\}^\omega$ is defined by $\mathsf{squares}(n) = 1$ iff $n+1$ is a square number. The sequence is morphic: it can be obtained by iterating the morphism $a \mapsto a001$, $0 \mapsto 0$, $1 \mapsto 001$ on the starting letter a, and applying the coding $a \mapsto 1$, $0 \mapsto 0$ and $1 \mapsto 1$ to the limit word.

We show that $\mathsf{squares}$ is not mix-automatic.

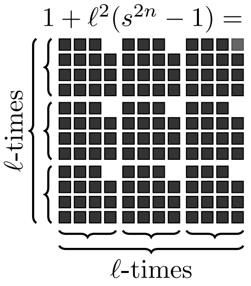
Lemma 20. *Let $\ell, s \in \mathbb{N}$ be such that $\ell, s > 1$. Then there exists a number $n \in \mathbb{N}$ such that $1 + \ell^2(s^n - 1)$ is not a square number.*

Proof. Let $\ell, s \in \mathbb{N}$ be such that $\ell, s > 1$. Let k be large enough to ensure $\ell < 2s^k$. Then $(\ell s^k - 1)^2 = \ell^2 s^{2k} - 2\ell s^k + 1 < 1 + \ell^2(s^{2k} - 1) < (\ell s^k)^2$ follows, which for $n = 2k$ traps $1 + \ell^2(s^n - 1)$ in between consecutive squares.

Alternatively, a geometrical rendering is the following. We view $s^{2n} - 1 = (s^n)^2 - 1$ as a square of $s^n \times s^n$ pieces of which one corner piece has been removed:



Then $1 + \ell^2(s^{2n} - 1)$ can be visualized as shown on the right. We have $\ell^2 - 1$ cut-out corner pieces. Due to these, $1 + \ell^2(s^{2n} - 1)$ is strictly less than a square $\ell s^n \times \ell s^n$. The next smaller square has size $(\ell s^n - 1) \times (\ell s^n - 1)$ and has precisely $(2\ell s^n - 1)$ less pieces than the larger square. By picking n large enough so that $\ell < 2s^n$, we achieve $\ell^2 - 1 < 2\ell s^n - 1$, and hence there are too many pieces for the next smaller square. □



Lemma 21. *The sequence* squares *is morphic but not mix-automatic.*

Proof. The morphic definition of squares is given above. For a contradiction, let us assume that the sequence would be mix-automatic. Then by Theorem 13, there exists $x : \Delta^\omega \to \mathbb{N}_{\geq 2}$ such that the x-kernel K of squares is finite. For every $n \in \mathbb{N}$ we define $w_n \in K$ and $k_n \in \mathbb{N}$ inductively as follows: $w_0 = \mathsf{squares}$ and $w_{n+1} = \pi_{0,k_n}(w_n)$ where $k_n = x(w_n)$. As K is finite, there exist $a, b \in \mathbb{N}$, $a < b$ such that $w_a = w_b$. We define $k = k_0 \cdot k_1 \cdots k_{a-1}$, and $\ell = k_a \cdot k_{a+1} \cdots k_{b-1}$. Then $w_a = \pi_{0,k}(\mathsf{squares})$ and $w_a = w_b = \pi_{0,\ell}(w_a)$, and in particular

$$\pi_{0,k}(\mathsf{squares}) = \pi_{0,\ell}(\pi_{0,\ell}(\pi_{0,k}(\mathsf{squares}))) = \pi_{0,k\ell^2}(\mathsf{squares})\,. \tag{3}$$

Thus (†) for all $n \in \mathbb{N}$, $kn + 1$ is a square if and only if $k\ell^2 n + 1$ is a square.

Let p be a prime that does not divide k, and hence is coprime to k. Then by Euler's theorem there exists $e \in \mathbb{N}$ such that $(p^2)^e \equiv 1 \pmod{k}$. Thus (∗) for every $m \in \mathbb{N}$, we have that $(p^{em})^2 = ((p^2)^e)^m \equiv 1 \pmod{k}$ and hence a square number of the form $kn + 1$ for some $n \in \mathbb{N}$.

We define $s = (p^2)^e$. Then an application of Lemma 20 yields that there exists $m \in \mathbb{N}$ such that $1 + \ell^2(s^m - 1)$ is not a square number. We have that $s^m = kn + 1$ for some $n \in \mathbb{N}$ by (∗). Thus $1 + \ell^2(s^m - 1) = 1 + \ell^2 kn$ is not a square while $1 + kn$ is. This contradicts (†). □

Theorem 22. *The class of morphic sequences is not contained in the class of mix-automatic sequences.*

6 Conclusions and Further Research

Mix-automatic sequences form a natural extension of the class of automatic sequences. While automatic sequences are generated by DFAOs, mix-automatic sequences are generated by DFAOs with state-dependent input alphabets. These automata read number representations $d_n d_{n-1} \cdots d_0$ where the base of a digit d_k depends on the value of the lower-significance digits $d_{k-1} \cdots d_0$.

The results of this paper can be summarized as follows:

(i) A characterization of mix-automatic sequences via a generalization of the concept of k-kernel (by which automatic sequences can be characterized).

(ii) For every polynomial φ there is a mix-automatic sequence whose subword complexity exceeds φ. As a consequence there are mix-automatic sequences that are not morphic, since morphic sequences have quadratic subword complexity at most.

(iii) A morphic sequence that is not mix-automatic, showing that the class of morphic sequences is not contained in the class of mix-automatic sequences.

All of these concepts are very recent, and many interesting questions remain. We highlight three particularly intriguing, and challenging questions:

(1) (J.-P. Allouche) Characterize the intersection of mix-automatic and morphic sequences. (Note that at least all automatic sequences are in.)

(2) Is the following problem decidable: Given two mix-DFAOs, do they generate the same sequence?

(3) Can Cobham's Theorem (below) be generalized to mix-automatic sequences?

Cobham's Theorem ([3]). Let $k, \ell \geq 2$ be multiplicatively independent (i.e., $k^a \neq \ell^b$, for all $a, b > 0$), and let $w \in \Delta^\omega$ be both k- and ℓ-automatic. Then w is ultimately periodic.

In order to generalize this theorem to mix-automatic sequences, one could look for a suitable notion of multiplicative independence for base determiners. Recall that base determiners are themselves finite automata with output.

References

1. Allouche, J.P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press, New York (2003)
2. Berstel, J.: Transductions and Context-Free Languages. Teubner (1979)
3. Cobham, A.: On the Base-Dependence of Sets of Numbers Recognizable by Finite Automata. Mathematical Systems Theory 3(2), 186–192 (1969)
4. Cobham, A.: Uniform Tag Sequences. Theory of Computing Systems 6, 164–192 (1972)
5. Ehrenfeucht, A., Lee, K.P., Rozenberg, G.: Subword Complexity of Various Classes of Deterministic Languages without Interaction. Theoretical Computer Science 1, 59–75 (1975)
6. Grabmayer, C., Endrullis, J., Hendriks, D., Klop, J.W., Moss, L.S.: Automatic Sequences and Zip-Specifications. In: Proc. Symp. on Logic in Computer Science (LICS 2012), pp. 335–344. IEEE Computer Society (2012)

7. Grabmayer, C., Endrullis, J., Hendriks, D., Klop, J.W., Moss, L.S.: Automatic Sequences and Zip-Specifications. Tech. Rep. 1201.3251, arXiv (2012), `http://arxiv.org/abs/1201.3251v1`
8. Knuth, D.E.: The Art of Computer Programming, vol. II: Seminumerical Algorithms, 2nd edn. Addison-Wesley (1981)
9. Morse, M., Hedlund, G.A.: Symbolic Dynamics. American Journal of Mathematics 60, 815–866 (1938)
10. Rigo, M.: Generalization of Automatic Sequences for Numeration Systems on a Regular Language. Theoretical Computer Science 244(1-2), 271–281 (2000)
11. Rigo, M., Maes, A.: More on Generalized Automatic Sequences. Journal of Automata, Languages and Combinatorics 7(3), 351–376 (2002)

A Multivariate Analysis of Some DFA Problems*

Henning Fernau[1], Pinar Heggernes[2], and Yngve Villanger[2]

[1] Department of Informatics, University of Trier, Germany
fernau@uni-trier.de

[2] Department of Informatics, University of Bergen, Norway
{pinar.heggernes,yngve.villanger}@ii.uib.no

Abstract. We initiate a multivariate analysis of two well-known NP-hard decision problems on DFAs: the problem of finding a short synchronizing word and that of finding a DFA on few states consistent with a given sample of the intended language and its complement. For both problems, we study natural parameterizations and classify them with the tools provided by Parameterized Complexity. Somewhat surprisingly, in both cases, rather simple FPT algorithms can be shown to be optimal, mostly assuming the (Strong) Exponential Time Hypothesis.

Keywords: Deterministic finite automata, NP-hard decision problems, synchronizing word, consistency problem.

1 Introduction

Multivariate analysis of computationally hard problems [16] tries to answer the question what actually makes a problem hard by systematically considering so-called natural parameters that can be singled out in an instance or in some target structure. In problems dealing with finite automata, such parameters could be the size of the input alphabet, or the number of states. For instance, if some hardness reduction produces or requires automata with large input alphabets, then this proof does not reveal much if only binary input alphabets are of interest. Parameterized Complexity offers tools to tell if hardness result could be expected when fixing, say, the alphabet size. In other words, we target the question what aspects of our problem cause it to become hard. As the possible choices of parameters are very abundant, we consider our paper rather as the starting point of this line of research within the theory of finite automata. Only limited multivariate analysis research has been undertaken so far on finite automata problems, NFA minimization being one exception [5], Mealy machines with census requirements another one [18], offering ample ground to work on.

In this paper, we study the parameterized complexity of two problems related to finite automata: the problem of finding a shortest synchronizing word in a deterministic finite automaton (DFA) and that of finding the smallest DFA consistent with a given sample consisting of positive and of negative examples of the intended language. Both problems have a long history, dating back to the very

* This work is supported by the Norwegian and the German Research Councils.

beginning of automata theory, and both questions have found many practical applications.

The Synchronizing Word (SW) problem is the following one: Given a DFA $A = (S, I, \delta, s_0, F)$ with state set S, input alphabet I, transition function $\delta : S \times I \to S$, initial state s_0 and set of final states F, together with some integer $k \geq 0$, decide if there exists a synchronizing word of length at most k for A. Here, a *synchronizing word* is a string $x \in I^*$ such that there exists some state $s_f \in S$ such that, for any start state $s \in S$, $\delta^*(s, x) = s_f$. Hence, a synchronizing word enables to reset an automaton to some well-defined state, wherever it may start. Therefore, it is also known as a *reset sequence* and also under many other different names. This notion and several related ones that we are going to discuss draw their practical motivation from testing circuits and automata, cf. [26,27,29].

Eppstein showed [15] that SW is NP-complete. Later, Berlinkov proved in [4] that the related optimization problem MIN-SW cannot belong to APX under some complexity assumptions. Walker [33] observed that Eppstein's reduction not only works when starting from 3-SAT, but also when using SAT. This will be useful for our purposes.

We show that SW is W[2]-hard when parameterized by the natural parameter k. This provides an alternative proof of the mentioned NP-hardness result. As our reduction is from Hitting Set, it also shows that MIN-SW cannot be approximated even up to some logarithmic factor depending on the size of the input alphabet [27,1]. This is not the same as Berlinkov's result, as he focuses on small alphabet sizes. It would be interesting to know if SW with parameter k actually belongs to W[2]. Otherwise, it might be one of the few natural problems known to be placed in higher levels of the W-hierarchy, cf. the discussions in [8].

The related combinatorial questions are nicely reported and reviewed in [29,32]. The most important question in that area is settling or disproving Černý's conjecture [7] that each t-state DFA that has a synchronizing word has also one of length at most $\frac{t(t-1)}{2}$. Currently best upper bounds are of size cubic in t, the record holder being [31].

As our second problem on automata, we consider the DFA Consistency problem. Here, the input consists in an alphabet I, two finite disjoint sets of words $X^+, X^- \subseteq I^*$, and an integer t. The question is to decide if there exists some DFA A with at most t states such that $X^+ \subseteq L(A)$ and $X^- \cap L(A) = \emptyset$. This problem was extensively studied in the area of Algorithmic Learning Theory, especially in Grammatical Inference, as it is central to the model of *Learning in the Limit*, initiated by Gold's seminal work [21]. As the problem was shown to be NP-hard [2,22,23] and even hard to approximate [28], several heuristics and translations to other problems were proposed. In our context, reductions to coloring problems seem to be most relevant, see [10]. We can underline our approach with a quotation from [23, p. 139]: *Alternative proofs of the hardness of the consistency problem would be of help, in order to better understand what is really hard.* This can be seen as a quest for a multivariate analysis for DFA Consistency, as we commence in this paper.

Notice that DFA Consistency can be seen as an implementation of Occam's razor in the sense that the shortest explanation (in terms of DFA) for the given sample is aimed at. This principle can also lead to computationally hard problems in the context of regular languages if only positive examples are given, taking into account questions concerning the representability or coding of the sample words. This kind of question was investigated in [17] from the viewpoint of Parameterized Complexity. Clearly, the consistency problem can be also asked for other types of automata and grammars. As long as the universal language I^* has a simple representation in the corresponding class, the consistency problem is trivial if $X^- = \emptyset$. However, there are also interesting classes of languages where this is not the case. For instance, it has been shown in [9] that this type of consistency problem is W[2]-hard for a whole range of categorial grammar families. Further Parameterized Complexity results for algorithmic learning problems can be found in [3,13,30]. There are also not so many papers dealing with Parameterized Complexity classification of questions on finite automata, cf. [18,34] in this context.

In this extended abstract, we had to omit most of the proofs. A long version of the paper can be obtained from the authors on request.

2 Preliminaries

A graph $G = (V, E)$ is undirected and unweighted, with vertex set V and edge set E. Given a subset $X \subseteq V$, the subgraph of G induced by X is denoted by $G[X]$. A vertex subset X is an *independent set* if $G[X]$ has no edges. A partition of V into $V_1, V_2, \ldots, V_r$ is called a *proper r-coloring* if $G[V_i]$ is an independent set for $1 \leq i \leq r$.

A *deterministic finite automaton (DFA) A* is a tuple (S, I, δ, s_0, F), where S is the set of states, I is the input alphabet, $\delta : S \times I \to S$ is the transition function, s_0 is the initial state, and F is the set of final states. We will use $t = |S|$.

For an introduction to the by now well established field of Parameterized Complexity, we refer the reader to the textbook [14]. Here we give a short and informal overview. A decision problem is said to be a *parameterized* problem if its input can be partitioned into a *main part J* and a *parameter P*. A parameterized problem with main input size $|J|$ and parameter size $|P|$ is said to be *fixed-parameter tractable* (FPT) if it can be solved by an algorithm with running time $O^*(f(|P|))$, where f is a computable function depending only on P and not on J, and the O^*-notation suppresses all factors that are polynomial in $|J|$. It is well-known that a parameterized problem is FPT if and only if it has a *kernel*, meaning that there is a polynomial time algorithm that produces an equivalent instance J' of size $|J'| \in O(g(|P|))$, where g is again a function depending only on P. If g is a polynomial function, then the problem is said to admit a *polynomial kernel.* Whether or not a fixed-parameter tractable problem admits a polynomial kernel is a broad subfield of Parameterized Complexity.

In the same way as NP-hardness of a decision problem indicates that we cannot expect a polynomial time algorithm, there exists a hierarchy of complexity classes above FPT, and showing that a parameterized problem is hard for any of these classes makes it unlikely to be FPT. The main classes are: FPT $\subseteq$ W[1] $\subseteq$ W[2] $\subseteq \ldots \subseteq$ W[P] $\subset$ XP, where XP is the class of parameterized problems that are solvable in time $O(|J|^{h(|P|)})$ for some function h. Consequently, if the problem is NP-hard when the parameter size is bounded by a constant, then it is not even likely to belong to XP.

Next we define some well-known problems and complexity theoretical assumptions, which will be useful for proving hardness results and lower bounds.

Problem: r-SAT
Input: A boolean CNF formula ϕ on n variables and m clauses, where each clause contains at most r literals.
Question: Is there a truth assignment that satisfies ϕ?

If there is no bound on the number of literals that a clause can contain, then we simply refer to the problem as SAT.

One common way to argue for the unlikeness of a subexponential algorithm is to use the *Exponential Time Hypothesis* (ETH). By the observation that each variable is used at least once, and by the Sparsification Lemma [25], ETH can be expanded to:

Exponential Time Hypothesis (ETH)[25]: There is a positive real s such that 3-SAT instances on n variables and m clauses cannot be solved in time $2^{sn}(n+m)^{O(1)}$.
Most useful is the following corollary: There is a real $s' > 0$ such that 3-SAT instances on m clauses cannot be solved in time $2^{s'm}(n+m)^{O(1)}$.

A slightly stronger assumption is the following one.
Strong Exponential Time Hypothesis (SETH)[25][6]: SAT cannot be solved in time $2^{sn}(n+m)^{O(1)}$ for $s < 1$.

In order to argue for the unlikeliness of polynomial kernels we use:

Proposition 1 ([19]). SAT*, parameterized by the number* n *of variables, does not have a kernel that is polynomial in* n *unless* NP $\subseteq$ coNP/poly*.*

We make use of the following NP-complete problems in our reductions [14,20].

Problem: r-Coloring
Input: A graph G on n vertices and m edges.
Question: Is there a proper r-coloring of G?

Problem: Hitting set
Input: A family $\mathcal{F}$ of sets over a universe $\mathcal{U}$ and an integer k.
Parameter: k
Question: Does there exist a set $\mathcal{S} \subset \mathcal{U}$ such that $|\mathcal{S}| \leq k$ and $F \cap \mathcal{S} \neq \emptyset$ for each $F \in \mathcal{F}$?

3 Shortest Synchronizing Word

We consider different parameterizations of the following problem.

Problem: Synchronizing Word (SW)
Input: A DFA $A = (S, I, \delta, s_0, F)$ and an integer k.
Question: Is there a k-synchronizing word for A?

The most natural parameter is of course k. As we will show, the problem is W[2]-hard with this parameter. We will consider other natural parameters as well; these are $t = |S|$ and $|I|$. Note that s_0 and F are simply irrelevant for SW, and $|\delta|$ is simply t times $|I|$. Table 1 summarizes the results of this section.

Table 1. A summary of the results of this section. In addition, we show that the two given running time upper bounds are tight in the sense that we cannot expect to solve SW in time $O^*(2^{o(t)})$ or in time $O^*((|I| - \epsilon)^k)$ for any $\epsilon > 0$.

Parameter	Parameterized Complexity	Polynomial Kernel
k	W[2]-hard	—
$\|I\|$	NP-complete for $\|I\| = 2$	—
k and $\|I\|$	FPT with running time $O^*(\|I\|^k)$	Not unless NP $\subseteq$ coNP/poly
t	FPT with running time $O^*(2^t)$	Open

For the first hardness result, we first need the following lemma.

Lemma 2. *Given a* Hitting Set *instance with family $\mathcal{F}$ and universe $\mathcal{U}$, a DFA $A = (S, I, \delta, s_0, F)$ can be constructed in time $O(|\mathcal{F}||\mathcal{U}|)$, such that $|S| = |\mathcal{F}| + k + 1$, $|I| = |\mathcal{U}|$, and A has a k-synchronizing word iff $\mathcal{F}$ has a hitting set of size k.*

Theorem 3. Synchronizing Word *is W[2]-hard, parameterized with k.*

If we instead parameterize SW with $|I|$, we obtain that the problem is not even in XP. For that result, we first need the following.

Proposition 4 ([15],[33]). *Given a* SAT *formula ϕ with n variables and m clauses, a DFA $A = (S, I, \delta, s_0, F)$ can be constructed in $O(nm)$ time, such that $|S| = nm + m + 1$, $|I| = 2$, and A has an n-synchronizing word if and only if ϕ has a satisfying truth assignment.*

Theorem 5. Synchronizing Word *is NP-complete when $|I| = 2$.*

As neither parameter k nor parameter $|I|$ is useful for fixed-parameter tractability, a natural next step is to use both k and $|I|$ as a combined parameter.

Theorem 6. Synchronizing Word *is FPT when parameterized with $|I|$ and k; it can be solved in time $O^*(|I|^k)$.*

The above result is straight-forward, and one could hope for an improvement or a polynomial kernel for SW when parameterized with both k and $|I|$. Interestingly, no such improvement seems likely, as we show next.

Lemma 7. *Given a CNF formula ϕ with n variables and m clauses, a DFA $A = (S, I, \delta, s_0, F)$ can be constructed in $O(nm)$ time, such that $|S| = n + m + 1$, $|I| = 2n$, and A has an n-synchronizing word if and only if ϕ is satisfiable.*

Proof. Let $V = \{x_1, \ldots, x_n\}$ be a set of variables in ϕ. Let $C = \{c_1, \ldots, c_m\}$ be the set of clauses in ϕ. We assume, w.l.o.g., that no variable occurs twice in any clause. The alphabet I contains $2n$ symbols x_i and $\overline{x_i}$ for $1 \leq i \leq n$, corresponding to the literals in the formula. We have the following $n + m + 1$ states:
Variable states q_i, $1 \leq i \leq n$; clause states c_j , $1 \leq j \leq m$; one sink state s.
The transitions are as follows:

1. $\delta(q_i, x_i) = \delta(q_i, \overline{x_i}) = q_{i+1}$, with $q_{n+1} = s$; $\delta(q_i, x_j) = \delta(q_i, \overline{x_j}) = q_i$ if $j \neq i$.
2. $\delta(c_j, l) = c_j$ for literal l if $l \notin c_j$. $\delta(c_j, l) = s$ for literal l if $l \in c_j$.
3. $\delta(s, l) = s$ for any literal l.

Notice that, as there are no transitions leading from the sink state to any other state, the state in which the synchronizing word (if it exists) must end is clear, it must be s. Any synchronizing word must be of length at least n as this is the length of the shortest path from q_1 to s. More precisely, any synchronizing word must be of the form described by the following regular expression:

$$(x_1 \cup \overline{x_1})^+ (x_2 \cup \overline{x_2})^+ \cdots (x_v \cup \overline{x_n})^+ (x_1 \cup \overline{x_1} \cup x_2 \cup \overline{x_2} \cup \cdots \cup x_n \cup \overline{x_n})^*.$$

This word should reflect the variable assignment. Namely, if there is a synchronizing word $\sigma = l_1 \cdots l_n$ of length n, then we can read off a variable assignment $\Phi : V \to \{0, 1\}$ as follows:

$$\Phi(x_i) = \begin{cases} 1, & \text{if } l_i = x_i \\ 0, & \text{if } l_i = \overline{x_i} \end{cases}$$

As σ leads into s in particular for each state c_j, this means that each clause c_j is satisfied by construction. The converse is similarly seen. □

Theorem 8. Synchronizing Word *cannot be solved in time $O^*(2^{o(t)})$ unless ETH fails.*

Proof. We start by using Lemma 7 to reduce a 3-SAT instance on n variables and m clauses to a SW instance $(A = (S, I, \delta, s_0, F), k)$ where $t = n + m + 1$, $|I| = 2n$, and $k = n$. If an algorithm existed that solved any SW instance in $O^*(2^{o(t)})$, then it would also solve 3-SAT in $O^*(2^{o(n+m)})$ time, contradicting ETH. □

Theorem 9. Synchronizing Word *does not have a polynomial kernel when parameterized with both k and $|I|$ unless* NP $\subseteq$ coNP/poly.

Proof. By Proposition 4 any CNF formula can be reduced to a SW instance $(A = (S, I, \delta, s_0, F), k)$ with $|I| = 2$ and $k = n$. If there existed a polynomial algorithm that produced an equivalent instance of size polynomial in $k, |I|$, this would mean that the number of states is reduced. As SW is NP-complete, there exists a polynomial time reduction back to a CNF formula with n' variables and m' clauses where $n' + m'$ is polynomial in $k, |I|$. This would imply that the number of clauses in this new CNF formula is bounded by a polynomial in n, and it is thus a polynomial kernel for SAT when parameterized by the number of variables n. By Proposition 1 this implies that $NP \subseteq coNP/poly$. □

Finally we turn our attention to parameter t. Again, there is a straight-forward FPT algorithm which is best possible.

Theorem 10 ([29]). Synchronizing Word *is FPT when parameterized with t; it can be solved in time $O^*(2^t)$.*

Theorem 11. Synchronizing Word *cannot be solved in time $O^*((|I| - \epsilon)^k)$ for any $\epsilon > 0$ unless SETH fails.*

Table 2. The table summarizes the results of this section. In addition we show that the parameter combination $(t, |I|)$ does not admit a polynomial kernel.

Parameter	Parameterized Complexity	Running time lower bound
t	NP-complete for $t = 2$	-
ℓ	NP-complete for $\ell = 2$	-
$\|I\|$	NP-complete for $\|I\| = 2$	-
c	Open	-
t, ℓ	NP-complete for $t\ell = 6$	-
$t, \|I\|$	FPT, running time $O^*(t^{t\|I\|})$	No $O^*(t^{o(t\|I\|)})$-time algorithm under ETH
t, c	Open	-
t, c, ℓ	FPT, running time $O^*(t^{c\ell})$	No $O^*(t^{o(c\ell)})$-time algorithm under ETH

4 DFA Consistency

In this section, we consider various parameterizations of the following problem:

Problem: DFA Consistency
Input: An alphabet I, two finite disjoint sets $X^+, X^- \subseteq I^*$, and an integer t
Question: Is there a DFA A with at most t states such that X^+ is accepted by A and X^- is rejected by A?

The natural parameters we work with here are the number of states t in the target DFA, the alphabet size $|I|$, the number of words $c = |X^+ \cup X^-|$, and the maximum length ℓ of any of the words in $X^+ \cup X^-$, i.e., $\max\{|\sigma| \mid \sigma \in X^+ \cup X^-\}$. The results of this section are summarized in Table 2. Notice that for the special case where $c = 2$ this problem is called the Separating Word Problem (for DFA), a recent overview can be found in [12].

Lemma 12. *Given a CNF formula φ with n variables and m clauses, an instance of the* DFA Consistency *problem can be constructed in time $O(nm)$, where $t = 2$, $|I| = 3n + 1$, $\ell = 2n$, and $c = 6n + m + 3$, such that there is a DFA on these parameters that distinguishes X^+ and X^- iff φ is satisfiable.*

A similar statement was shown by D. Angluin in 1989, but never published.

Lemma 13. *Let $G = (V, E)$ on n vertices and m edges be an instance of* 3-Coloring. *Then an instance of* DFA Consistency *can be constructed in time $O(n + m)$, where $t = 3$, $|I| = n + m$, $\ell = 2$, $c = 2m$, and such that there exists a DFA on these parameters that distinguishes X^+ and X^- iff G is 3-colorable.*

Proof. Let us first construct the DFA Consistency instance and then argue that it is a YES instance if and only if the 3-Coloring instance is a YES instance. We start by setting $t = 3$ and $I = V \cup E$, which leaves the definition of X^+ and X^-. Let $v_1, \ldots, v_n$ be an arbitrary numbering of the vertices in V. The sets of words X^+ and X^- are now constructed as follows:

– $X^+ = \{v_i e \mid e = v_i v_j \in E, i < j\}$;
– $X^- = \{v_j e \mid e = v_i v_j \in E, i < j\}$.

This completes the construction of the DFA Consistency instance.

Let us now argue for the equivalence of the two instances. For the first direction we assume that there exists a DFA $A = (S, I, \delta, s_0, F)$ on three states and alphabet $I = V \cup E$ that accept X^+ and rejects X^-. Let s_0, s_1, s_2 be the states of A and let v_i be contained in V_q for $0 \leq q \leq 2$ if $\delta(s_1, v_i) = s_q$. This gives us a partitioning V_0, V_1, V_2 of V. Our objective will now be to argue that V_q is an independent set in G for $0 \leq q \leq 2$. On the contrary, let $e = v_i v_j \in E$ where $i < j$ be an edge such that $v_i, v_j \in V_q$. From the construction of X^+ and X^- it is clear that set X^+ contains word $v_i e$ and set X^- contains $v_j e$. As the only difference between these two words is the first symbol and one word is accepted and the other one is rejected, it is clear that different states are reached by reading v_i and v_j from the start state s_0. Thus, either v_i or v_j is not contained in V_q and the contradiction is obtained.

For the second direction assume that there is a partitioning V_0, V_1, V_2 of V such that V_q is an independent set for $0 \leq q \leq 2$. Name the three states s_0, s_1, s_2 and let s_0 be the start state and s_1 the only accepting state. Function δ is now defined as follows:

1. $\delta(s_1, v_i) = s_q$, for $v_i \in V_q$ where $0 \leq q \leq 2$;
2. $\delta(s_q, v_i v_j) = s_1$ for $0 \leq q \leq 2$ and $i < j$;
3. $\delta(s_q, v_i v_j) = s_2$ for $0 \leq q \leq 2$ and $i > j$;

It is not hard to verify that all words in X^+ are accepted and all words in X^- are rejected. □

Lemma 14. *(also see [23]) Given a CNF formula φ with n variables and m clauses, an instance of the* DFA Consistency *problem can be constructed in* *time $O((n+m)^2)$, where $t = n+m+1$, $|I| = 2$, $\ell = m+n$, and $c = 5m+n+4$, such that there exists a DFA on these parameters that distinguishes X^+ and X^- iff φ is satisfiable.*

Theorem 15. DFA Consistency *cannot be solved in time $O^*(t^{o(t|I|)})$ unless ETH fails.*

Proof. Through the standard reduction from 3-SAT to 3-Coloring it follows that that 3-Coloring instance on n vertices and m edges can not be solved in time $O^*(2^{o(n+m)})$ unless ETH fails.

By the reduction of Lemma 13 we get an instance of DFA Consistency where $t = 3, |I| = n+m, \ell = 2$, and $c = 2m$. Any algorithm for DFA Consistency solving it in $O^*(t^{o(t|I|)})$ time will also solve 3-Coloring in $O^*(2^{o(n+m)})$ and ETH will fail. □

Theorem 16. DFA Consistency *does not have a polynomial kernel when parameterized with both t and $|I|$ unless* NP $\subseteq$ coNP/poly.

Proof. By Lemma 12 any CNF formula can be reduced to a a DFA Consistency instance where $t = 2$, $|I| = 3n+1$, $\ell = 2n$, and $c = 6n+m+3$ in polynomial time. If there existed a polynomial algorithm that produced an equivalent instance of size polynomial in $t, |I|$, this would mean that the number of words in $X^+ \cup X^-$ is reduced. As DFA Consistency is NP-complete, there exists a polynomial time reduction back to a SAT instance with n' variables and m' clauses where $n' + m'$ is polynomial in $t, |I|$. This would imply that the number of clauses in this CNF formula is bounded by a polynomial in n, and it is thus a polynomial kernel for SAT when parameterized by the number of variables. By Proposition 1 this implies that *$NP \subseteq coNP/poly$.* □

Next we turn to parameter combination (t, c, ℓ), which again gives a trivial FPT algorithm whose running time seems unlikely to be improvable.

Theorem 17. DFA Consistency *is FPT when parameterized with t, c, and ℓ; it can be solved in time $O^*(t^{c\ell})$.*

Theorem 18. DFA Consistency *cannot be solved in time $O^*(t^{o(c\ell)})$ unless ETH fails.*

Proof. Through the standard reduction from 3-SAT to 3-Coloring it follows that that 3-Coloring instance on n vertices and m edges can not be solved in time $O^*(2^{o(n+m)})$ unless ETH fails. By the reduction of Lemma 13 we get an instance of the DFA Consistency problem where $t = 3, |I| = n+m, \ell = 2$, and $c = 2m$. Any algorithm for the DFA Consistency problem solving the problem in $O^*(t^{o(c\ell)})$ time will also solve the 3-Coloring problem in $O^*(2^{o(m)})$ and ETH will fail. □

We end this section by turning our attention to parameter c. Could it be that DFA Consistency is NP-hard when $t = 2$ and c is bounded by a constant? We are able to answer this question partially with the below positive result.

Theorem 19. DFA Consistency *can be solved in polynomial time when $t = 2$ and $c = 2$.*

Table 3. A summary of the results on Q-SYNCHRONIZING WORD

Parameter	Parameterized Complexity
t	FPT with running time $O^*(2^t)$
$\|I\|$	PSPACE-complete for $\|I\| = 2$
k	W[2]-hard
k and $\|I\|$	FPT with running time $O^*(\|I\|^k)$
$\|Q\|$ and k	W[1]-hard
$\|Q\|$ and $\|I\|$	W[t]-hard for all t

5 Other Related Problems

The two core problems we investigated so far have quite a number of interesting variants for which several of our results carry over. We focus on SW-variants, often computationally harder than SW.

Based on the assumption that some partial information on the current state of a DFA might be known, formalized by a set of states Q, the Q-SYNCHRONIZING WORD (Q-SW) problem was introduced. In this problem we are only interested in finding a word x, $|x| \leq k$, that synchronizes all states from Q, i.e., $|\delta^*(Q, x)| = 1$. From [34] and the reduction from DFA INTERSECTION NONEMPTINESS given in [29] that shows PSPACE-hardness of this problem, we can immediately deduce the last two rows of Table 3. The only technical problem is that in the parameterized analogue DFA INTERSECTION, the length parameter m is an exact bound, while the length parameter k is an upper bound. However, by adding a sequence of m "new" states starting from some Q-state s_0, we can enforce the constructed DFA to have a word of length at least $m + 1$ as its shortest Q-synchronizing word. The reduction given in [29] will increase the word length by one.

Our parameterized complexity results for parameters t, $|I|$, and k transfer from SYNCHRONIZING WORD to this more general setting. Table 3 summarizes our results.

We also considered related problems on Mealy machines. For reasons of space, we only mention that finding short homing sequences leads to complexity results similar to synchronizing words, while finding short distinguishing sequences is more complex, simmilar to Q-synchronizing words.

6 Conclusion and Questions for Future Research

With this paper, we started some first steps in the multivariate analysis of several DFA (and Mealy machine) problems. Several questions emerge.

- Does SYNCHRONIZING WORD have a polynomial kernel with parameter t?
- Is DFA CONSISTENCY FPT when parameterized with c or with c and t?
- Does DFA CONSISTENCY have a polynomial kernel with parameter (t, c, ℓ)?

- There are other natural variants of DFA CONSISTENCY. Angluin showed [2] that REGULAR EXPRESSION CONSISTENCY is even hard for regular expressions of a very simple structure, without any nested Kleene stars, which sits very low in the famous star height hierarchy, see [11]. In view of the fact that for many applications, regular expressions are considered as important as DFAs, this could give an interesting line of research.
- What could be further natural parameters for problems on regular languages? Discovering these as possible sources of hardness could be a very fruitful line of research for both problem classes that we considered in this paper. Thoughts from the classical theory of Formal Languages could become very helpful, for instance, from Descriptional Complexity [24].

References

1. Alon, N., Moshkovitz, D., Safra, S.: Algorithmic construction of sets for k-restrictions. ACM Transactions on Algorithms 2(2), 153–177 (2006)
2. Angluin, D.: On the complexity of minimum inference of regular sets. Information and Control (now Information and Computation) 39, 337–350 (1978)
3. Arvind, V., Köbler, J., Lindner, W.: Parameterized learnability of juntas. Theoretical Computer Science 410(47-49), 4928–4936 (2009)
4. Berlinkov, M.V.: Approximating the Minimum Length of Synchronizing Words Is Hard. In: Ablayev, F., Mayr, E.W. (eds.) CSR 2010. LNCS, vol. 6072, pp. 37–47. Springer, Heidelberg (2010)
5. Björklund, H., Martens, W.: The tractability frontier for NFA minimization. Journal of Computer and System Sciences 78(1), 198–210 (2012)
6. Calabro, C., Impagliazzo, R., Paturi, R.: The Complexity of Satisfiability of Small Depth Circuits. In: Chen, J., Fomin, F.V. (eds.) IWPEC 2009. LNCS, vol. 5917, pp. 75–85. Springer, Heidelberg (2009)
7. Černý, J.: Poznámka k homogénnym experimentom s konečnými automatmi. Matematicko-fyzikálny Časopis 14(3), 208–216 (1964)
8. Chen, J., Zhang, F.: On product covering in 3-tier supply chain models: Natural complete problems for $W[3]$ and $W[4]$. Theoretical Computer Science 363(3), 278–288 (2006)
9. Costa Florêncio, C., Fernau, H.: On families of categorial grammars of bounded value, their learnability and related complexity questions. Theoretical Computer Science 452, 21–38 (2012)
10. Costa Florêncio, C., Verwer, S.: Regular Inference as Vertex Coloring. In: Bshouty, N.H., Stoltz, G., Vayatis, N., Zeugmann, T. (eds.) ALT 2012. LNCS (LNAI), vol. 7568, pp. 81–95. Springer, Heidelberg (2012)
11. Dejean, F., Schützenberger, M.P.: On a question of Eggan. Information and Control (now Information and Computation) 9(1), 23–25 (1966)
12. Demaine, E.D., Eisenstat, S., Shallit, J., Wilson, D.A.: Remarks on Separating Words. In: Holzer, M. (ed.) DCFS 2011. LNCS, vol. 6808, pp. 147–157. Springer, Heidelberg (2011)
13. Downey, R.G., Evans, P.A., Fellows, M.R.: Parameterized learning complexity. In: Proc. Sixth Annual ACM Conference on Computational Learning Theory, COLT, pp. 51–57. ACM Press (1993)
14. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer (1999)

15. Eppstein, D.: Reset sequences for monotonic automata. SIAM Journal on Computing 19(3), 500–510 (1990)
16. Fellows, M.: Towards Fully Multivariate Algorithmics: Some New Results and Directions in Parameter Ecology. In: Fiala, J., Kratochvíl, J., Miller, M. (eds.) IWOCA 2009. LNCS, vol. 5874, pp. 2–10. Springer, Heidelberg (2009)
17. Fellows, M.R., Fernau, H.: Facility location problems: A parameterized view. Discrete Applied Mathematics 159, 1118–1130 (2011)
18. Fellows, M.R., Gaspers, S., Rosamond, F.A.: Parameterizing by the number of numbers. Theory of Computing Systems 50(4), 675–693 (2012)
19. Fortnow, L., Santhanam, R.: Infeasibility of instance compression and succinct PCPs for NP. In: Dwork, C. (ed.) ACM Symposium on Theory of Computing, STOC, pp. 133–142. ACM (2008)
20. Garey, M.R., Johnson, D.S.: Computers and Intractability. Freeman, New York (1979)
21. Gold, E.M.: Language identification in the limit. Information and Control (now Information and Computation) 10, 447–474 (1967)
22. Gold, E.M.: Complexity of automaton identification from given data. Information and Control (now Information and Computation) 37, 302–320 (1978)
23. de la Higuera, C.: Grammatical inference. Learning automata and grammars. Cambridge University Press (2010)
24. Holzer, M., Kutrib, M.: Descriptional and computational complexity of finite automata - a survey. Information and Computation 209(3), 456–470 (2011)
25. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity? Journal of Computer and System Sciences 63(4), 512–530 (2001)
26. Kohavi, Z.: Switching and Finite Automata Theory. McGraw-Hill (1970)
27. Lee, D., Yannakakis, M.: Testing finite state machines: State identification and verification. IEEE Transactions on Computers 43, 306–320 (1994)
28. Pitt, L., Warmuth, M.K.: The minimum consistent DFA problem cannot be approximated within any polynomial. Journal of the ACM 40, 95–142 (1993)
29. Sandberg, S.: Homing and Synchronizing Sequences. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 5–33. Springer, Heidelberg (2005)
30. Stephan, F., Yoshinaka, R., Zeugmann, T.: On the parameterised complexity of learning patterns. In: Gelenbe, E., Lent, R., Sakellari, G. (eds.) Computer and Information Sciences II - 26th International Symposium on Computer and Information Sciences, ISCIS, pp. 277–281. Springer (2011)
31. Trahtman, A.N.: Modifying the Upper Bound on the Length of Minimal Synchronizing Word. In: Owe, O., Steffen, M., Telle, J.A. (eds.) FCT 2011. LNCS, vol. 6914, pp. 173–180. Springer, Heidelberg (2011)
32. 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)
33. Walker, P.J.: Synchronizing Automata and a Conjecture of Černý. Ph.D. thesis, University of Manchester, UK (2008)
34. Wareham, H.T.: The Parameterized Complexity of Intersection and Composition Operations on Sets of Finite-State Automata. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 302–310. Springer, Heidelberg (2001)

On the Size Complexity of Deterministic Frequency Automata[⋆]

Rūsiņš Freivalds[1,2], Thomas Zeugmann[2], and Grant R. Pogosyan[3]

[1] Institute of Mathematics and Computer Science, University of Latvia, Raiņa bulvāris 29, Riga, LV-1459, Latvia
Rusins.Freivalds@mii.lu.lv

[2] Division of Computer Science, Hokkaido University, N-14, W-9, Sapporo 060-0814, Japan
thomas@ist.hokudai.ac.jp

[3] Division of Natural Sciences, International Christian University, Mitaka, Tokyo 181-0015, Japan
grant@icu.ac.jp

Abstract. Austinat, Diekert, Hertrampf, and Petersen [2] proved that every language L that is (m, n)-recognizable by a deterministic frequency automaton such that $m > n/2$ can be recognized by a deterministic finite automaton as well. First, the size of deterministic frequency automata and of deterministic finite automata recognizing the same language is compared. Then approximations of a language are considered, where a language L' is called an *approximation* of a language L if L' differs from L in only a finite number of strings. We prove that if a deterministic frequency automaton has k states and (m, n)-recognizes a language L, where $m > n/2$, then there is a language L' approximating L such that L' can be recognized by a deterministic finite automaton with no more than k states.

Austinat *et al.* [2] also proved that every language L over a single-letter alphabet that is $(1, n)$-recognizable by a deterministic frequency automaton can be recognized by a deterministic finite automaton. For languages over a single-letter alphabet we show that if a deterministic frequency automaton has k states and $(1, n)$-recognizes a language L then there is a language L' approximating L such that L' can be recognized by a deterministic finite automaton with no more that k states. However, there are approximations such that our bound is much higher, i.e., $k!$.

1 Introduction

The notion of frequency computation was introduced by Rose [18] as an attempt to have an absolutely deterministic mechanism with properties similar to probabilistic algorithms. The definition was as follows. Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ denote

[⋆] The research was supported by Grant No. 09.1570 from the Latvian Council of Science and the Invitation Fellowship for Research in Japan S12052 by the Japan Society for the Promotion of Science.

the set of all natural numbers. A function $f\colon \mathbb{N} \to \mathbb{N}$ is (m, n)-computable, where $1 \le m \le n$, $m, n \in \mathbb{N}$, iff there exists a recursive function $R\colon \mathbb{N}^n \to \mathbb{N}^n$ such that, for all n-tuples $(x_1, \cdots, x_n) \in \mathbb{N}^n$ of mutually distinct natural numbers,

$$\text{card}\{i \mid (R(x_1, \cdots, x_n))_i = f(x_i)\ ,\ 1 \le i \le n\} \ge m\ ,$$

where $(R(x_1, \cdots, x_n))_i$ denotes the ith component of $R(x_1, \cdots, x_n)$.

McNaughton [16] cites in his survey a problem (posed by Myhill) whether f has to be recursive if m is close to n. This problem was answered by Trakhtenbrot [20] by showing that f is recursive whenever $2m > n$. On the other hand, Trakhtenbrot [20] proved that, if $2m = n$ then nonrecursive functions can be (m, n)-computed. Kinber [14, 13] extended the research by considering frequency enumeration of sets. The class of (m, n)-computable sets equals the class of recursive sets if and only if $2m > n$. The notion of frequency computation can be extended to other models of computation. Frequency computation in polynomial time was discussed in full detail by Hinrichs and Wechsung [11].

For resource bounded computations, the behavior of frequency computability is completely different: for example, whenever $n' - m' > n - m$, it is known that under any reasonable resource bound there are sets which are (m', n')-computable, but not (m, n)-computable. However, scaling down to finite automata, the analogue of Trakhtenbrot's [20] result holds again: the class of languages (m, n)-recognizable by deterministic frequency automata equals the class of regular languages if and only if $2m > n$ (cf. Austinat *et al.* [2]). Conversely, as shown by Austinat *et al.* [2], for $2m \le n$, the class of languages (m, n)-recognizable by deterministic frequency automata is uncountable for a two-letter alphabet. A stronger result concerning sets separable by finite automata was claimed by Kinber [13], and this result would imply the results mentioned above as a corollary. However, as shown by Tantau [19], who gave a counter-example, Kinber's [13] Theorem 3 does not hold.

When restricted to a one-letter alphabet, then every (m, n)-recognizable language is regular. This was shown by Kinber [14] and also by Austinat *et al.* [2].

Frequency computations became increasingly popular when relations between frequency computation and computation with a small number of queries was discovered [1, 2, 3, 4, 5, 8, 10, 15].

2 Deterministic Frequency Automata

For finite automata the definition of frequency computation is not so obvious. First, let us fix the necessary notations. We assume familiarity with finite automata theory, cf., e.g., Hopcroft and Ullman [12]. Let Σ be any finite alphabet, and let Σ^* be the free monoid over Σ. Every subset $L \subseteq \Sigma^*$ is said to be a *language*. The elements of Σ^* are called *strings*, and we use $|x|$ to denote the *length of a string* $x \in \Sigma^*$. By $\chi_L\colon \Sigma^* \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$, we denote the *characteristic function* of L, i.e., for all $x \in \Sigma^*$ we set

$$\chi_L(x) = \begin{cases} 1, & \text{if } x \in L\ ; \\ 0, & \text{if } x \notin L\ . \end{cases}$$

To define deterministic frequency automata we extend the notion of a deterministic finite automaton as follows (cf. Austinat *et al.* [2]).

Let $\mathcal{A} = [Q, \Sigma, \#, \delta, q_0, \tau, n]$, where $n \in \mathbb{N}$, $n \geq 1$, is a number, Q is a finite set of states, q_0 is the initial state, Σ is a finite alphabet and $\#$ is a new symbol such that $\# \notin \Sigma$. The mapping $\delta\colon Q \times (\Sigma \cup \{\#\})^n \to Q$ is the transition function, and we call $\tau\colon Q \to \mathbb{B}^n$ the type of state. The type of state is used for the output. We refer to $\mathcal{A}$ as *deterministic frequency automaton.*

Next, we formally describe the behavior of a deterministic frequency automaton $\mathcal{A}$. Let $n \in \mathbb{N}$, $n \geq 1$, and let $x = (x_1, \ldots, x_n) \in (\Sigma^*)^n$ be an input vector. We define $|x| = \max\{|x_i| \mid 1 \leq i \leq n\}$, and $q \circ x = \delta^*(q, (x_1\#^{\ell_1}, \ldots, x_n\#^{\ell_n}))$, where $\delta^*\colon Q \times ((\Sigma \cup \{\#\})^n)^*$ is the usual extension of δ on n-tuples of strings, and $\ell_i = |x| - |x_i|$ for all $1 \leq i \leq n$. Then the output of $\mathcal{A}$ is defined to be the type $\tau(q_0 \circ x)$. We refer to such an automaton as n-DFA for short.

A language $L \subseteq \Sigma^*$ is said to be (m, n)-*recognized* by an n-DFA $\mathcal{A}$ iff for each n-tuple $(x_1, \ldots, x_n) \in (\Sigma^*)^n$ of pairwise distinct strings the tuples $\tau(q_0 \circ x)$ and $(\chi_L(x_1), \ldots, \chi_L(x_n))$ coincide on at least m components. A language $L \subseteq \Sigma^*$ is called (m, n)-recognizable iff there is an n-DFA $\mathcal{A}$ that (m, n)-recognizes L.

Frequency computation is not much similar to probabilistic computation. The advantages of probabilistic algorithms over deterministic ones are based on the effect that at some moments the algorithm has a choice of several possible continuations of the computation process but there is no information which one suits better. For example, if the language is $L_{3,5} = \{1^n \mid n \text{ is divisible by 3 or 5}\}$ then a probabilistic algorithm has a choice: whether to test divisibility by 3 or divisibility by 5.

Frequency algorithms have no such option. Nonetheless, frequency automata can have size complexity advantages over deterministic automata as well. To see this, consider the language $L_{2015} \subseteq \{1\}^*$ defined as

$$L_{2015} = \{1^n \mid n = 2015\} . \tag{1}$$

A deterministic finite automaton recognizing this language needs to have 2016 states. On the other hand, there is a 1-state 100-DFA $\mathcal{A}$ that $(99, 100)$-recognizes the language L_{2015}. The 100-DFA $\mathcal{A}$ rejects all 100-tuples, i.e., it always outputs $\tau(q_0 \circ x) = (b_1, \ldots, b_{100}) \in \mathbb{B}^{100}$, where $b_i = 0$ for all $i = 1, \ldots, 100$. But nonetheless $\mathcal{A}$ does $(99, 100)$-recognize the language L_{2015}, since it can only be wrong on at most one of the 100 strings in any 100-tuple given as input.

This idea can be easily extended. Consider $L_{2015,2158} \subseteq \{1\}^*$ defined as

$$L_{2015,2158} = \{1^n \mid n = 2015 \text{ or } n = 2158\} . \tag{2}$$

Then there exists a 1-state 100-DFA $\mathcal{A}$ such that $\mathcal{A}$ does $(98, 100)$-recognize the language $L_{2015,2158}$. Again, $\mathcal{A}$ rejects all 100-tuples, but nonetheless recognizes the language $L_{2015,2158}$.

Maybe, the only advantage of deterministic frequency automata over deterministic finite automata is to save size by producing errors on a constant number of fixed input words? Not at all, some nonregular and even nonrecursive languages can be recognized by deterministic frequency automata.

Theorem 1 (Austinat *et al.* [2]). *There exists a nonrecursive language that is* $(1, 2)$*-recognizable by a* 2*-DFA.*

3 Size Complexity

In this section we study the size complexity of deterministic frequency automata and compare it to the size of ordinary deterministic automata recognizing the same language or an approximation of it.

First, we extend the example shown in (1). This directly yields the following theorem showing that the advantage of deterministic frequency automata with respect to their size complexity over deterministic finite automata can be arbitrarily large.

Theorem 2. *For every* $s \in \mathbb{N}$*,* $s \geq 1$*, there is a language* L_s *such that, for every* $n \geq 1$ *there is an* n*-DFA* $\mathcal{A}$ *having one state that* $(n-1, n)$*-recognizes* L_s*, but every deterministic finite automaton recognizing* L_s *needs at least* s *states.*

Proof. Let $s \geq 1$ be arbitrarily fixed and let L_s be any language defined as

$$L_s = \{1^m \mid m = m_0\} \ ,$$

where m_0 is a number such that every deterministic finite automaton needs at least s states to recognize the language L_s. The desired n-DFA $\mathcal{A}$ can then be easily defined such that $Q = \{q_0\}$ and such that for all n-tuples of strings $x \in (\{1\}^*)^n$ the mapping $\tau(q_0 \circ x)$ returns the n-tuple containing only zeros. Hence, L_s is $(n-1, n)$-recognized by $\mathcal{A}$, since $\mathcal{A}$ rejects all input strings. But an error can happen only once, i.e., if 1^{m_0} is part of the input. □

Clearly, Theorem 2 can be easily generalized along the lines of the example shown in (2). So the more interesting question is whether or not there is always a (huge) gap in the size of deterministic frequency automata and deterministic finite automata provided they accept roughly the same language. Here by "roughly" we mean that we allow the deterministic finite automaton to accept an approximation of the language accepted by the corresponding deterministic frequency automaton.

Austinat *et al.* [2] proved that in the case $m > n/2$ every language (m, n)-recognized by an n-DFA is also recognizable by a deterministic finite automaton. There were no size estimates of the deterministic frequency automata and the deterministic finite automata, respectively, in [2] but a careful optimization of the construction given in [2] proves the following theorem.

Theorem 3. *Let any pair* (m, n)*, where* $m > n/2$*, be arbitrarily fixed, and let* $\mathcal{A}$ *be any* n*-DFA having* k *states. If there is a language* L *which is* (m, n)*-recognized by the* n*-DFA* $\mathcal{A}$ *then there exists a language* L' *differing from* L *only in a finite number of strings such that* L' *can be recognized by a deterministic finite automaton with* 2^{k+3} *states.*

It is well-known that for nondeterministic finite automata and probabilistic 1-way automata the size gap to deterministic finite automata recognizing the same language is exponential [17, 6, 7, 9]. But now we consider approximations and compare n-DFA and deterministic finite automata. Our next theorem shows that the gap expressed in Theorem 3 between the sizes of deterministic frequency automata and deterministic finite automata is not necessary provided $|\Sigma| \geq 2$.

Theorem 4. *Let any pair (m, n), where $m > n/2$, be arbitrarily fixed, and let $\mathcal{A}$ be any n-DFA having k states. If there is a language L which is (m, n)-recognized by the n-DFA $\mathcal{A}$ then there exists a language L' differing from L only in a finite number of strings such that L' can be recognized by a deterministic finite automaton with k states.*

Proof. This proof is nonconstructive. This means that we do not present an effective construction how to transform a program for (m, n)-recognition of a language into a program for deterministic recognition of the same language. Instead we show that such a transformation can be done using a finite amount of additional information and we show that such an additional information cannot fail to exist but we do not show how to obtain such additional information effectively. Since $|\Sigma| \geq 2$, we assume without loss of generality that $\{0, 1\} \subseteq \Sigma$.

By $[\alpha_1, \alpha_2, \cdots, \alpha_k/\beta_1, \beta_2, \cdots, \beta_k]$, where $\alpha_i, \beta_i \in \{0, 1\}$, $1 \leq i \leq k$, we denote the set of all strings $x \in \Sigma^*$ such that

$$\begin{aligned}
\chi_L(x\alpha_1) &= \beta_1 \ , \\
\chi_L(x\alpha_1\alpha_2) &= \beta_2 \ , \\
&\cdots \\
\chi_L(x\alpha_1\alpha_2\cdots\alpha_k) &= \beta_k \ .
\end{aligned}$$

We start to describe a noneffective "construction" of a tree denoted by S. This tree is defined inductively. In principle, the tree might be infinite but we prove below that the "construction" results in a finite tree. We will use the resulting tree as the finite additional information about the given frequency automaton.

- There is a single vertex of the zero level in the tree (called the root). All the strings $x \in \Sigma^*$ are assigned to it.
- If the vertex of the p-th level is already in the tree with the set

  $$[\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p]$$

  used as label to it, we "consider" whether the sets

  $$\begin{aligned}
  &[\alpha_1, \alpha_2, \cdots, \alpha_p, 0/\beta_1, \beta_2, \cdots, \beta_p, 0] \quad \text{and} \\
  &[\alpha_1, \alpha_2, \cdots, \alpha_p, 0/\beta_1, \beta_2, \cdots, \beta_p, 1]
  \end{aligned}$$

  are infinite. If the two sets are infinite then we add two $(p + 1)$-th level vertices to the tree and label them by the sets

  $$\begin{aligned}
  &[\alpha_1, \alpha_2, \cdots, \alpha_p, 0/\beta_1, \beta_2, \cdots, \beta_p, 0] \quad \text{and} \\
  &[\alpha_1, \alpha_2, \cdots, \alpha_p, 0/\beta_1, \beta_2, \cdots, \beta_p, 1] \ ,
  \end{aligned}$$

respectively. We say that these two vertices

$$\begin{aligned}&[\alpha_1, \alpha_2, \cdots, \alpha_p, 0/\beta_1, \beta_2, \cdots, \beta_p, 0] \quad \text{and}\\ &[\alpha_1, \alpha_2, \cdots, \alpha_p, 0/\beta_1, \beta_2, \cdots, \beta_p, 1]\end{aligned}$$

are *strictly higher* than the vertex v labeled by

$$[\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p] \tag{3}$$

If at least one of these two sets is finite or empty then no vertex is added in the result of this "consideration." In this case, we continue as follows. If the two sets

$$\begin{aligned}&[\alpha_1, \alpha_2, \cdots, \alpha_p, 1/\beta_1, \beta_2, \cdots, \beta_p, 0] \quad \text{and}\\ &[\alpha_1, \alpha_2, \cdots, \alpha_p, 1/\beta_1, \beta_2, \cdots, \beta_p, 1]\end{aligned}$$

are infinite, then we add two $(p+1)$-th level vertices to the tree S and label them by the sets

$$\begin{aligned}&[\alpha_1, \alpha_2, \cdots, \alpha_p, 1/\beta_1, \beta_2, \cdots, \beta_p, 0] \quad \text{and}\\ &[\alpha_1, \alpha_2, \cdots, \alpha_p, 1/\beta_1, \beta_2, \cdots, \beta_p, 1]\ ,\end{aligned}$$

respectively. Again, we say that these two vertices are strictly higher than v. If at least one of these two sets is finite or empty then no vertex is added.

Notice two properties of the sets assigned to the vertices of the tree S.

(1) If a vertex

$$[\alpha_1, \alpha_2, \cdots, \alpha_{p+1}/\beta_1, \beta_2, \cdots, \beta_{p+1}]$$

is strictly higher than

$$[\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p]$$

then the following inclusion holds:

$$[\alpha_1, \alpha_2, \cdots, \alpha_{p+1}/\beta_1, \beta_2, \cdots, \beta_{p+1}] \subseteq [\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p]\ .$$

(2) If two vertices

$$\begin{aligned}&[\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p] \quad \text{and}\\ &[\gamma_1, \gamma_2, \cdots, \gamma_r/\delta_1, \delta_2, \cdots, \delta_r]\end{aligned}$$

are distinct vertices in the tree S then there exist strings

$$\begin{aligned}&x \in [\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p] \quad \text{and}\\ &y \in [\gamma_1, \gamma_2, \cdots, \gamma_r/\delta_1, \delta_2, \cdots, \delta_r]\end{aligned}$$

and a string $z \in \Sigma^*$ such that $(xz \in L) \not\Leftrightarrow (yz \in L)$.

Property (1) is an immediate consequence of the construction.

The second property is proved separately for the following two cases.

Case (a). One of the vertices is strictly higher than the other one, e.g., $p > r$.

By the definition of strictly higher (cf. (3)) we then have $\alpha_i = \gamma_i$ and $\beta_i = \delta_i$ for $i = 1, \ldots, r$ and so, by Property (1) and the transitivity of set inclusion, $[\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p] \subseteq [\alpha_1, \alpha_2, \cdots, \alpha_r/\beta_1, \beta_2, \cdots, \beta_r]$. Then we take any arbitrarily chosen string $x \in [\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p]$, and any string $y \in [\alpha_1, \alpha_2, \cdots, \alpha_r/\beta_1, \beta_2, \cdots, \beta_r] \setminus [\alpha_1, \alpha_2, \cdots, \alpha_r, \alpha_{r+1}/\beta_1, \beta_2, \cdots, \beta_r, \beta_{r+1}]$, and define $z = \alpha_1 \cdots \alpha_r \alpha_{r+1}$, i.e., the concatenation of $\alpha_1, \ldots, \alpha_{r+1}$. By construction x, y, and z clearly satisfy Property (2).

The subcase $p < r$ is handled *mutatis mutandis.*

Case (b). None of the vertices $[\alpha_1, \cdots, \alpha_p/\beta_1, \cdots, \beta_p]$ and $[\gamma_1, \cdots, \gamma_r/\delta_1, \cdots, \delta_r]$ is strictly higher than the other one.

Let j be the least number less than $\min\{p, r\}$ such that $\alpha_i = \gamma_i$ and $\beta_i = \delta_i$ for all $i = 1, \ldots, j$. Note that we allow $j = 0$ to denote the case that the highest such vertex is the root, i.e., the vertex of level zero.

Consequently, we then have that $\alpha_{j+1} \neq \gamma_{j+1}$ or $\beta_{j+1} \neq \delta_{j+1}$. But by construction we know that $\alpha_{j+1} \neq \gamma_{j+1}$ cannot occur. Therefore, we know that $\beta_{j+1} \neq \delta_{j+1}$ must hold, i.e., $\gamma_{j+1} = \overline{\beta}_{j+1}$, where $\overline{b}$ denotes the logical negation of $b \in \{0, 1\}$. This in turn implies that

$$[\alpha_1, \ldots, \alpha_{j+1}/\beta_1, \ldots, \beta_{j+1}] \cap [\alpha_1, \ldots, \alpha_{j+1}/\beta_1, \ldots, \overline{\beta}_{j+1}] = \emptyset \ , \tag{4}$$

i.e., these two sets partition $[\alpha_1, \ldots, \alpha_j/\beta_1, \ldots, \beta_j]$. So by Property (1) and Equality (4) we also have

$$[\gamma_1, \cdots, \gamma_r/\delta_1, \cdots, \delta_r] \cap [\alpha_1, \ldots, \alpha_{j+1}/\beta_1, \ldots, \beta_{j+1}] = \emptyset \ . \tag{5}$$

Therefore, we can take any string $x \in [\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p]$, and any string $y \in [\gamma_1, \cdots, \gamma_r/\delta_1, \cdots, \delta_r]$. Furthermore, we set $z = \alpha_1 \cdots \alpha_{j+1}$.

So, if $j = 0$ then x and y are as above, and $z = \alpha_1$. Thus, $\chi_L(x\alpha_1) = \beta_1$ and $\chi_L(y\alpha_1) \neq \beta_1$. In the general case that $j > 0$ we directly obtain that $\chi_L(xz) = \beta_{j+1}$ and $\chi_L(yz) \neq \beta_{j+1}$ (cf. (5)), and Property (2) is shown.

Claim: The tree S has at most k vertices.

Suppose that the tree S has at least $(k+1)$ vertices. Take an n-tuple of pairwise distinct strings from each of the sets, and, again nonconstructively, appropriate $z^1, \ldots, z^{k+1}$ (such that Property (2) will be applicable). Denote the resulting tuples by $(x_1^1 z^1, \cdots, x_n^1 z^1)$, $(x_1^2 z^2, \cdots, x_n^2 z^2)$, $\ldots$, $(x_1^{k+1} z^{k+1}, \cdots, x_n^{k+1} z^{k+1})$.

The n-DFA $\mathcal{A}$ has only k states but we have taken $(k+1)$ many n-tuples of strings. Hence there are two distinct values i and j such that, after reading $(x_1^i z^i \cdots, x_n^i z^i)$ and $(x_1^j z^j \cdots, x_n^j z^j)$, the n-DFA $\mathcal{A}$ is in the same state. So, by the choice of the z^ℓ, $\ell = 1, \ldots, k+1$, we know that $z^i = z^j =: z$ and that Property (2) is applicable. Since we also have $m > n/2$, if the n-tuple corresponding to the output on $(x_1^i z \cdots, x_n^i z)$ contains no less than m correct values then the n-DFA cannot produce an n-tuple as output which contains m many correct values on the input $(x_1^j z, \cdots, x_n^j z)$. This is a contradiction, and the claim is shown.

Now we add one more level to the tree S in order to construct a new tree S'. Denote the highest level of S by s, We have already shown that $s \leq k-1$. Every vertex of the highest level and only these vertices have the following property. If the vertex is

$$[\alpha_1, \alpha_2, \cdots, \alpha_p / \beta_1, \beta_2, \cdots, \beta_p]$$

then there may exist a $\gamma \in \Sigma$ such that the sets

$$[\alpha_1, \alpha_2, \cdots, \alpha_p, \gamma / \beta_1, \beta_2, \cdots, \beta_p, 0] \quad \text{and}$$
$$[\alpha_1, \alpha_2, \cdots, \alpha_p, \gamma / \beta_1, \beta_2, \cdots, \beta_p, 1]$$

are infinite. In this case we add

$$[\alpha_1, \alpha_2, \cdots, \alpha_p, \gamma / \beta_1, \beta_2, \cdots, \beta_p, 0] \quad \text{and}$$
$$[\alpha_1, \alpha_2, \cdots, \alpha_p, \gamma / \beta_1, \beta_2, \cdots, \beta_p, 1]$$

as new vertices of the k-th level over the vertex

$$[\alpha_1, \alpha_2, \cdots, \alpha_p / \beta_1, \beta_2, \cdots, \beta_p] .$$

Each newly added vertex either corresponds to some state of the n-DFA $\mathcal{A}$ that is already related to another vertex of S' or it corresponds to a state of $\mathcal{A}$ that yet has no vertex in S'. We transform S' into a graph which is no longer a tree by identifying vertices that correspond to the same state of the n-DFA $\mathcal{A}$. We continue adding new and new vertices in the same style. For every vertex that has been added in the process of transforming the tree S we try to add new vertices of a higher level but we identify them with vertices already constructed if no new state of the n-DFA $\mathcal{A}$ is employed. It is easy to see that the obtained graph (we call it S'') has no more than k vertices.

To construct an equivalent deterministic finite automaton we notice that in the construction above we distinguished between "infinitely many strings" and "a finite number of strings." Let d exceed the length of all considered "a finite number of strings." Then every string of length no less than d falls in one or several of the sets which are names of vertices in S''. Consider unions of such sets. We say that a union of the sets which are names of vertices in S'' is *consistent* if it is a union of type

$$[\alpha_1/\beta_1] \cup [\alpha_1, \alpha_2/\beta_1, \beta_2] \cup \cdots \cup [\alpha_1, \alpha_2, \cdots, \alpha_p/\beta_1, \beta_2, \cdots, \beta_p].$$

We say that a union is *complete* if it is not possible to add any other set which is a name of a vertex in S'' to this union.

It is easy to see that every consistent and complete union has incorporated a vertex of the upper-most level which is not present at any other consistent and complete union. Hence the number of consistent and complete unions does not exceed the number of vertices in the graph S'', i.e., it does not exceed k.

Consider a deterministic finite automaton that has states numbered by the consistent and complete unions of vertices of the tree S'. The initial state (for the empty input string) is the vertex of the level zero. The state numbered $[\alpha_1, \alpha_2, \cdots, \alpha_j / \beta_1, \beta_2, \cdots, \beta_j]$ is accepting if and only if $\beta_j = 1$. Since $m > n/2$, it is possible to construct the program for the deterministic finite automaton counting to which state the transition should take place. For strings that have a length exceeding d this automaton is equivalent to the given n-DFA $\mathcal{A}$ and its number of states does not exceed k. □

The formulation of Theorem 4 may seem to be over-complicated. Why another language is considered? It is well-known that a language differing from a regular languages only in a finite number of strings is itself regular. However, the number of states may be influenced very much if we neglect this finite number of strings (cf. Theorem 2, and Theorems 3 and 4, respectively).

Theorem 4 has a sensitive restriction: the considered deterministic frequency automata have parameters (m, n) with $m > n/2$. Looking at Theorem 1 we see that this restriction cannot be relaxed.

Therefore, we turn our attention to the unary case, i.e., only single-letter alphabets are allowed. Then the situation changes drastically, since the following theorem is known.

Theorem 5 (Austinat *et al.* [2], Kinber [13]). *Let any pair (m, n), where $0 < m \leq n$, be arbitrarily fixed, and let $\mathcal{A}$ be any n-DFA having k states. Then every language L over a single-letter alphabet that is (m, n)-recognized by the n-DFA $\mathcal{A}$ is also recognizable by a deterministic finite automaton.*

We complement Theorem 5 in terms of size complexity of the automata.

Theorem 6. *Let any pair (m, n), where $0 < m \leq n$, be arbitrarily fixed, and let $\mathcal{A}$ be any n-DFA having k states. If there is a language L over a single-letter alphabet Σ which is (m, n)-recognized by the n-DFA $\mathcal{A}$ then there exists a language $L' \subseteq \Sigma^*$ differing from L only in a finite number of strings such that L' can be recognized by a deterministic finite automaton with k states.*

Proof. Without loss of generality, let the single-letter alphabet $\Sigma = \{1\}$. If an n-DFA $\mathcal{A}$ does (m, n)-recognize a language $L \subseteq \Sigma^*$ then $\mathcal{A}$ $(1, n)$-recognizes L, too. Let Q be the set of states of $\mathcal{A}$. By assumption we know that $|Q| = k$. For all $q_b \in Q$ we consider the following sets of n-tuples of input strings:

$$\begin{aligned} T_b = \{(1^{m_1}, 1^{m_2}, \cdots, 1^{m_n}) \mid \mathcal{A} & \text{ after reading} \\ & (1^{m_1}, 1^{m_2}, \cdots, 1^{m_n}) \text{ enters state } q_b\} , \end{aligned}$$

where all $m_i \geq 1$, i.e., all m_i are positive natural numbers. Note that the class T_b is labeled by the index b of the state q_b but not by the n-tuple of input strings.

Let $\mathcal{T}$ be the collection $\mathcal{T} = \{T_b \mid T_b \text{ is infinite }\}$. We define a relation between the sets T_b in $\mathcal{T}$. Let T_b and T_c be any sets in $\mathcal{T}$. Then we say that

$(1^{m_1}, 1^{m_2}, \cdots, 1^{m_n}) \in T_b$ *precedes* $(1^{s_1}, 1^{s_2}, \cdots, 1^{s_n}) \in T_c$ if there exists a string 1^v such that

$$m_1 + v = s_1, m_2 + v = s_2, \cdots, m_n + v = s_n \ .$$

Precedence of n-tuples induces precedence of sets in $\mathcal{T}$. Since all the sets $T_b \in \mathcal{T}$ are infinite, for every pair (T_b, T_c), either T_b precedes T_c or T_c precedes T_b or T_b does not precede T_c and T_c does not precede T_b. The precedence relation divides the sets into equivalence classes corresponding to states b and c in the same cycle.

Let k be the number of the states in $\mathcal{A}$. Then each set $T_b \in \mathcal{T}$ contains an n-tuple $(1^{m_1}, 1^{m_2}, \cdots, 1^{m_n})$ such that $m_1 + m_2 + \cdots + m_n \leq n \cdot k$.

It may be possible that the same automaton $\mathcal{A}$ $(1, n)$-recognizes several languages. Let L be one of these languages. Then it is possible to make assertions about which components of the n outputs of $\mathcal{A}$ on a certain n-tuple of input strings are correct. By the definition of frequency computation, for every n-tuple of input strings at least one of the outputs is correct.

Assume that each equivalence class of the sets T_b is represented by one set from the equivalence class:

$$(1^{m_1}, 1^{m_2}, \cdots, 1^{m_n}),\ (1^{p_1}, 1^{p_2}, \cdots, 1^{p_n}),\ \cdots,\ (1^{s_1}, 1^{s_2}, \cdots, 1^{s_n}) \ .$$

Let these classes contain $t_m, t_p, \cdots, t_s$ sets T_b, respectively. Let N be the least common multiple of $t_m, t_p, \cdots, t_s$.

Assume that one of the equivalence classes is represented by n-tuples of input strings

$$\begin{gathered}
(1^{m_1}, 1^{m_2}, \cdots, 1^{m_n}) \ , \\
(1^{m_1+1}, 1^{m_2+1}, \cdots, 1^{m_n+1}) \ , \\
\cdots \\
(1^{m_1+t_m}, 1^{m_2+t_m}, \cdots, 1^{m_n+t_m}) \ .
\end{gathered}$$

Then by analyzing the outputs on these n-tuples

$$\begin{gathered}
(y_1^1, y_2^1, \cdots, y_n^1) \\
(y_1^2, y_2^2, \cdots, y_n^2) \\
\cdots \\
(y_1^{t_m}, y_2^{t_m}, \cdots, y_n^{t_m})
\end{gathered}$$

we can find one or several periodical sequences $f = \langle f(1), f(2), \cdots \rangle$ of elements $0, 1$ (any such sequence describes a language in a single-letter alphabet such that $f(n) = 1$ iff 1^n is in the language) with period t_m such that

$$\begin{gathered}
(f(m_1) = y_1^1) \vee (f(m_2) = y_2^1) \vee \cdots \vee (f(m_n) = y_n^1) \\
(f(m_1 + 1) = y_1^2) \vee (f(m_2 + 1) = y_2^2) \vee \cdots \vee (f(m_n + 1) = y_n^2) \\
\cdots \\
(f(m_1 + t_m) = y_1^{t_m}) \vee (f(m_2 + t_m) = y_2^{t_m}) \vee \cdots \vee (f(m_n + t_m) = y_n^{t_m}) \ .
\end{gathered}$$

The Chinese Remainder Theorem gives an algorithm of how to find such sequences. However, combining all these calculations give us only a sequence (or several sequences) with period $N \leq k!$. On the other hand, the states of the frequency automaton are periodically repeated with periods $t_m, t_p, \cdots, t_s$, respectively. The least common multiple of all these periods is N. Hence the language can be recognized by a deterministic finite automaton with N states.

There is no need to perform these calculations by a finite automaton. After performing these calculations we can reconstruct a full period of the length N and construct the program of the deterministic finite automaton.

However, a deeper analysis of the frequency automaton is needed, since our theorem promises a deterministic finite automaton with no more than k states, and not N states. Let L be a language $(1,n)$-recognized by the n-DFA $\mathcal{A}$. For each cycle

$$\begin{gathered}(1^{m_1}, 1^{m_2}, \cdots, 1^{m_n})\ , \\ (1^{m_1+1}, 1^{m_2+1}, \cdots, 1^{m_n+1})\ , \\ \cdots \\ (1^{m_1+t_m}, 1^{m_2+t_m}, \cdots, 1^{m_n+t_m})\ .\end{gathered}$$

of the frequency automaton we can say which outputs

$$\begin{gathered}(y_1^1, y_2^1, \cdots, y_n^1) \\ (y_1^2, y_2^2, \cdots, y_n^2) \\ \cdots \\ (y_1^{t_m}, y_2^{t_m}, \cdots, y_n^{t_m})\end{gathered}$$

are correct and which are not. For every cycle we establish whether for some $i \in \{1, 2, \cdots, t_m\}$ all the outputs

$$y_i^1, y_i^2, \cdots, y_i^{t_m}$$

are correct. If there is such a cycle and such an i then the i-th output of this cycle provides correct results for all sufficiently long input strings.

If such an i does not exist for all cycles then the n-DFA $\mathcal{A}$ does not recognize L correctly, because, by Chinese Remainder Theorem, there is an n-tuple of input strings such that all the outputs of $\mathcal{A}$ are incorrect. □

However, Theorem 6 does not hold for all approximations.

Theorem 7. *Let any number $n \in \mathbb{N}$, $n \geq 1$, be arbitrarily fixed, and let $\mathcal{A}$ be any n-DFA having k states. If there is a language L over a single-letter alphabet Σ which is $(1,n)$-recognized by the n-DFA $\mathcal{A}$ then there exists a language $L' \subseteq \Sigma^*$ differing from L only in a finite number of strings such that L' can be only recognized by a deterministic finite automaton which has at least $k!$ states.*

References

[1] Ablaev, F.M., Freivalds, R.: Why Sometimes Probabilistic Algorithms Can Be More Effective. In: Wiedermann, J., Gruska, J., Rovan, B. (eds.) MFCS 1986. LNCS, vol. 233, pp. 1–14. Springer, Heidelberg (1986)

[2] Austinat, H., Diekert, V., Hertrampf, U., Petersen, H.: Regular frequency computations. Theoretical Computer Science 330(1), 15–21 (2005)

[3] Beigel, R., Gasarch, W., Kinber, E.: Frequency computation and bounded queries. Theoretical Computer Science 163(1-2), 177–192 (1996)

[4] Case, J., Kaufmann, S., Kinber, E.B., Kummer, M.: Learning recursive functions from approximations. J. Comput. Syst. Sci. 55(1), 183–196 (1997)

[5] Degtev, A.N.: On (m, n)-computable sets. In: Moldavanskij, D.I. (ed.) Algebraic Systems, pp. 88–99. Ivanovo Gos. Universitet (1981) (in Russian)

[6] Freivalds, R.: Recognition of languages with high probability of different classes of automata. Doklady Akademii Nauk SSSR 239(1), 60–62 (1978) (in Russian)

[7] Freivalds, R.: On the growth of the number of states in result of the determinization of probabilistic finite automata. Avtomatika i Vychislitel'naya Tekhnika 3, 39–42 (1982) (in Russian)

[8] Freivalds, R.: Complexity of Probabilistic Versus Deterministic Automata. In: Bārzdiņš, J., Bjørner, D. (eds.) Baltic Computer Science. LNCS, vol. 502, pp. 565–613. Springer, Heidelberg (1991)

[9] Freivalds, R.: Non-constructive methods for finite probabilistic automata. International Journal of Foundations of Computer Science 19(3), 565–580 (2008)

[10] Harizanov, V., Kummer, M., Owings, J.: Frequency computations and the cardinality theorem. The Journal of Symbolic Logic 57(2) (1992)

[11] Hinrichs, M., Wechsung, G.: Time bounded frequency computations. Information and Computation 139(2), 234–257 (1997)

[12] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison–Wesley Publishing Company, Reading (1979)

[13] Kinber, E.B.: Frequency computations in finite automata. Cybernetics and Systems Analysis 12(2), 179–187 (1976)

[14] Kinber, E.B.: Frequency calculations of general recursive predicates and frequency enumerations of sets. Soviet Mathematics Doklady 13, 873–876 (1972)

[15] Kummer, M.: A proof of Beigel's cardinality conjecture. The Journal of Symbolic Logic 57(2), 677–681 (1992)

[16] McNaughton, R.: The theory of automata, a survey. Advances in Computers 2, 379–421 (1961)

[17] Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development 3(2), 114–125 (1959)

[18] Rose, G.F.: An extended notion of computability. In: International Congress for Logic, Methodology and Philosophy of Science, Stanford University, Stanford, California, August 24-September 2 (1960); Abstracts of contributed papers

[19] Tantau, T.: Towards a Cardinality Theorem for Finite Automata. In: Diks, K., Rytter, W. (eds.) MFCS 2002. LNCS, vol. 2420, pp. 625–636. Springer, Heidelberg (2002)

[20] Trakhtenbrot, B.A.: On the frequency computation of functions. Algebra i Logika 2(1), 25–32 (1964) (in Russian)

On the Number of Unbordered Factors

Daniel Goč, Hamoon Mousavi, and Jeffrey Shallit

School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada
{dgoc,hamoon.mousavihaji,shallit}@cs.uwaterloo.ca

Abstract. We illustrate a general technique for enumerating factors of k-automatic sequences by proving a conjecture on the number $f(n)$ of unbordered factors of the Thue-Morse sequence. We show that $f(n) \leq n$ for $n \geq 4$ and that $f(n) = n$ infinitely often. We also give examples of automatic sequences having exactly 2 unbordered factors of every length.

Keywords: Thue-Morse sequence, period-doubling sequence, linear representation, unbordered factor, k-regular sequence, automatic sequence.

1 Introduction

In this paper, we are concerned with certain factors of k-automatic sequences. Roughly speaking, a sequence $\mathbf{x} = a_0 a_1 a_2 \cdots$ over a finite alphabet Δ is said to be k-automatic if there exists a finite automaton that, on input n expressed in base k, reaches a state with output a_n. Automatic sequences were popularized by a celebrated paper of Cobham [3] and have been widely studied; see [1].

More precisely, let k be an integer ≥ 2, and set $\Sigma_k = \{0, 1, \ldots, k-1\}$. Let $M = (Q, \Sigma_k, \Delta, \delta, q_0, \tau)$ be a deterministic finite automaton with output (DFAO) with transition function $\delta : Q \times \Sigma_k \to Q$ and output function $\tau : Q \to \Delta$. Let $(n)_k$ denote the canonical base-k representation of n, without leading zeros, and starting with the most significant digit. Then we say that M generates the sequence $(a_n)_{n \geq 0}$ if $a_n = \tau(\delta(q_0, (n)_k))$ for all $n \geq 0$.

The prototypical example of a k-automatic sequence is the Thue-Morse sequence $\mathbf{t} = t_0 t_1 t_2 \cdots = 01101001 \cdots$, defined by the relations $t_0 = 0$ and $t_{2n} = t_n$, $t_{2n+1} = 1 - t_n$ for $n \geq 0$. It is generated by the DFAO below in Figure 1.

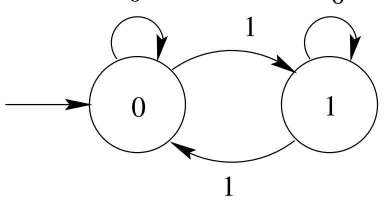


Fig. 1. A finite automaton generating the Thue-Morse sequence $\mathbf{t}$

A *factor* of the sequence $\mathbf{x}$ is a finite word of the form $a_i \cdots a_j$. A finite word w is said to be *bordered* if there is some finite nonempty word $x \neq w$ that is both a prefix and a suffix of w [12,11,6,13]. For example, the English word `ionization` is bordered, as it begins and ends with `ion`. Otherwise w is said to be *unbordered.*

Recently, there has been significant interest in the properties of unbordered factors; see, for example, [9,8,5,10]. In particular, Currie and Saari [4] studied the unbordered factors of the Thue-Morse word.

Currie and Saari [4] proved that if $n \not\equiv 1 \pmod 6$, then the Thue-Morse word has an unbordered factor of length n, but left it open to decide for which lengths congruent to 1 (mod 6) this property holds. This was solved in [7], where the following characterization is given:

Theorem 1. *The Thue-Morse sequence* $\mathbf{t}$ *has an unbordered factor of length* n *if and only if* $(n)_2 \notin 1(01^*0)^*10^*1$.

A harder problem is to come up with an expression for the number of unbordered factors of $\mathbf{t}$. In [2], the third author and co-authors made the following conjecture:

Conjecture 2. Let $f(n)$ denote the number of unbordered factors of length n in $\mathbf{t}$, the Thue-Morse sequence. Then f is given by $f(0) = 1$, $f(1) = 2$, $f(2) = 2$, and the system of recurrences

$$\begin{aligned}
f(4n+1) &= f(2n+1)\\
f(8n+2) &= f(2n+1) - 8f(4n) + f(4n+3) + 4f(8n)\\
f(8n+3) &= 2f(2n) - f(2n+1) + 5f(4n) + f(4n+2) - 3f(8n)\\
f(8n+4) &= -4f(4n) + 2f(4n+2) + 2f(8n)\\
f(8n+6) &= 2f(2n) - f(2n+1) + f(4n) + f(4n+2) + f(4n+3) - f(8n)\\
f(16n) &= -2f(4n) + 3f(8n) \qquad (1)\\
f(16n+7) &= -2f(2n) + f(2n+1) - 5f(4n) + f(4n+2) + 3f(8n)\\
f(16n+8) &= -8f(4n) + 4f(4n+2) + 4f(8n)\\
f(16n+15) &= -8f(4n) + 2f(4n+3) + 4f(8n) + f(8n+7).
\end{aligned}$$

for $n \geq 0$.

Although this conjecture may appear unmotivated, it is characteristic of the kinds of recurrences that naturally appear for k-regular sequences, and was obtained by computing a large number of values of f and then looking for possible linear relations among subsequences of the form $(f(2^i n + j))_{n \geq 0}$.

This system suffices to calculate f efficiently, in $O(\log n)$ arithmetic steps.

We now summarize the rest of the paper. In Section 2, we prove Conjecture 2. In Section 3, we discuss how to obtain relations like those above for a given k-regular sequence. In Section 4 we discuss the growth rate of f in detail. Finally, in Section 5, we give examples of other sequences with interesting numbers of unbordered factors.

2 Proof of the Conjecture

We now outline our computational proof of Conjecture 2.

First, we need a little notation. We extend the notion of canonical base-k representation of a single non-negative integer to tuples of such integers. For example, by $(m, n)_k$ we mean the unique word over the alphabet $\Sigma_k \times \Sigma_k$ such that the projection π_1 onto the first coordinate gives the base-k representation of m, and the projection π_2 onto the second co-ordinate gives the base-k representation of n, where the shorter representation is padded with leading 0's, if necessary, so that the representations have the same length. For example, $(43, 17)_2 = [1, 0][0, 1][1, 0][0, 1][1, 0][1, 1]$.

Proof. Step 1: Using the ideas in [7], we created an automaton A of 23 states that accepts the language L of all words $(n, i)_2$ such that there is a "novel" unbordered factor of length n in $\mathbf{t}$ beginning at position i. Here "novel" means that this factor does not previously appear in any position to the left. Thus, the number of such words with first component equal to $(n)_2$ equals $f(n)$, the number of unbordered factors of $\mathbf{t}$ of length n. This automaton is illustrated below in Figure 2 (rotated to fit the figure more clearly).

Step 2: Using the ideas in [2], we now know that f is a 2-regular sequence, with a "linear representation" that can be deduced from the structure of A. This gives matrices M_0, M_1 of dimension 23 and vectors v, w such that $f(n) = vM_{a_1} \cdots M_{a_i} w$ where $a_1 \cdots a_i$ is the base-2 representation of n, written with the most significant digit first. They are given below.

$$M_0 = \begin{bmatrix}
1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0
\end{bmatrix}$$

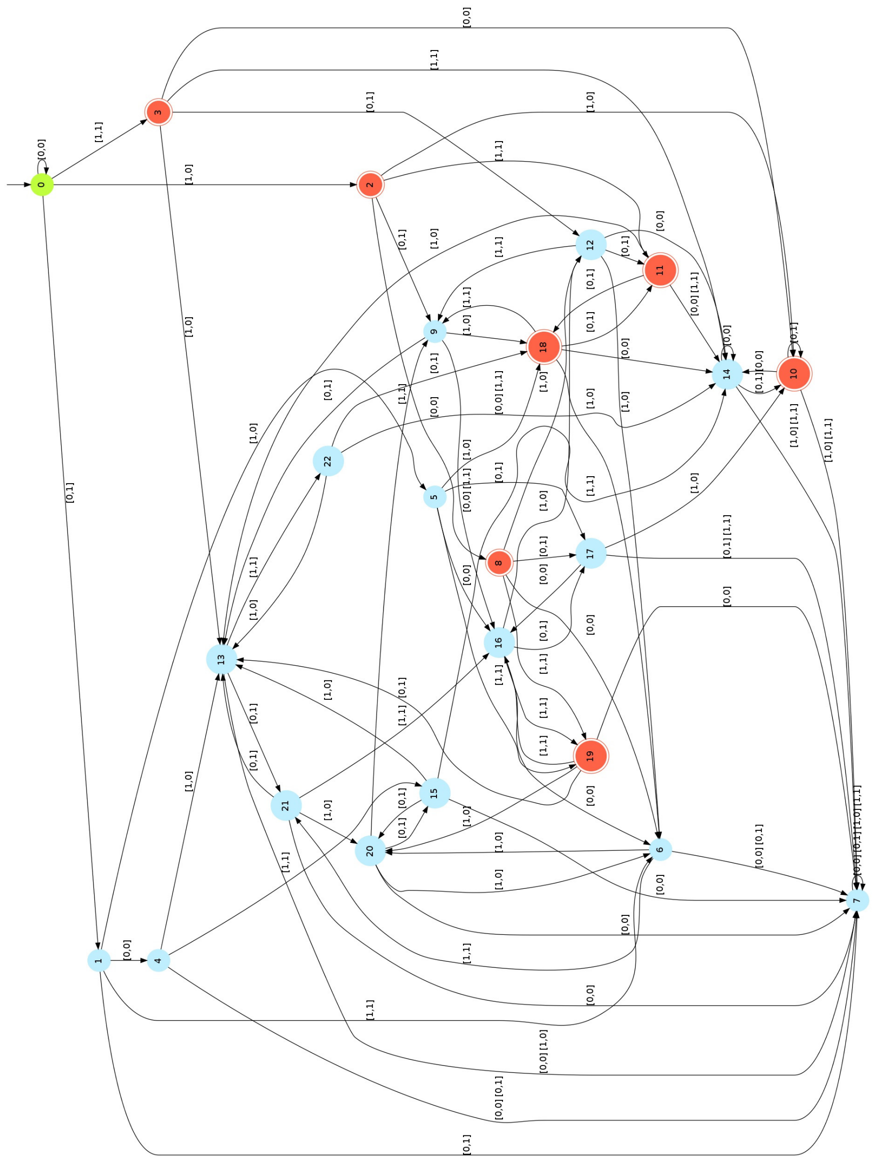


Fig. 2. Automaton accepting $(n, i)_2$ such that there is a novel unbordered factor of length n at position i of $\mathbf{t}$

$$M_1 = \begin{bmatrix}
0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}.$$

$$v = [1,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,1,0]$$
$$w = [0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]^T$$

Step 3: Now each of the identities in (1) corresponds to a certain identity in matrices. For example, the identity $f(16n) = -2f(4n) + 3f(8n)$ can be written as

$$vMM_0M_0M_0M_0w = -2vMM_0M_0w + 3vMM_0M_0M_0w, \tag{2}$$

where M is the matrix product corresponding to the base-2 expansion of n. More generally, we can think of M as some arbitrary product of the matrices M_0 and M_1, starting with at least one M_1; this corresponds to an arbitrary $n \geq 1$. We can think of M as a matrix of indeterminates. Then (2) represents an assertion about the entries of M which can be verified. Of course, the entries of M are not completely arbitrary, since they come about as M_1 times some product of M_0 and M_1. We can compute the (positive) transitive closure of $M_0 + M_1$ and then multiply on the left by M_1; the entries that have 0's will be 0 in any product of M_1 times a product of the matrices M_0 and M_1. Thus we can replace the corresponding indeterminates by 0, which makes verifying (2) easier.

Another approach, which is even simpler, is to consider vM in place of M. This reduces the number of entries it is required to check from d^2 to d, where d is the dimension of the matrices.

Step 4: Finally, we have to verify the identities for $n = 0$ and $n = 1$, which is easy.

We carried out this computation in Maple for the matrices M_0 and M_1 corresponding to A, which completes the proof. The Maple program can be downloaded from

`http://www.cs.uwaterloo.ca/~shallit/papers.html` .

3 Determining the Relations

The verification method of the previous section can be extended to a method to mechanically *find* the relations for *any* given k-regular sequence g (instead of guessing them and verifying them), given the linear representation of g.

Suppose we are given the linear representation of a k-regular sequence g, that is, vectors v, w and matrices $M_0, M_1, \ldots, M_{k-1}$ such that

$$g(n) = vM_{a_1}M_{a_2}\cdots M_{a_j}w,$$

where $a_1a_2\cdots a_j = (n)_k$.

Now let M be arbitrary and consider vM as a vector with variable entries, say $[a_1, a_2, \ldots, a_d]$. Successively compute vMM_yw for words y of length $0, 1, 2, \ldots$ over $\Sigma_k = \{0, 1, \ldots, k-1\}$; this will give an expression in terms of the variables $a_1, \ldots, a_d$. After at most $d+1$ such relations, we find an expression for vMM_yw for some y as a linear combination of previously computed expressions. When this happens, you no longer need to consider any expression having y as a suffix. Eventually the procedure halts, and this corresponds to a system of equations like that in (2).

Consider the following example. Let $k = 2$, $v = [6, 1]$, $w = [2, 4]^T$, and

$$M_0 = \begin{bmatrix} -3 & 1 \\ 1 & 4 \end{bmatrix}$$

$$M_1 = \begin{bmatrix} 0 & 2 \\ -3 & 1 \end{bmatrix}$$

Suppose M is some product of M_0 and M_1, and suppose $vM = [a, b]$. We find

$$\begin{aligned} vMw &= 2a + 4b \\ vMM_0w &= -2a + 18b \\ vMM_1w &= -8a - 2b \\ vMM_0M_0w &= 24a + 70b \\ vMM_1M_0w &= 36a + 24b \end{aligned}$$

and, solving the linear systems, we get

$$\begin{aligned} vMM_1w &= \frac{35}{11}vMw - \frac{9}{11}vM_0w \\ vMM_0M_0w &= 13vMw + vM_0w \\ vMM_1M_0w &= \frac{174}{11}vMw - \frac{24}{11}vM_0w. \end{aligned}$$

This gives us

$$\begin{aligned} g(2n+1) &= \frac{35}{11}g(n) + \frac{9}{11}g(2n) \\ g(4n) &= 13g(n) + g(2n) \\ g(4n+2) &= \frac{174}{11}g(n) - \frac{24}{11}g(2n) \end{aligned}$$

for $n \geq 1$.

4 The Growth Rate of $f(n)$

We now return to $f(n)$, the number of unbordered factors of $\mathbf{t}$ of length n. Here is a brief table of $f(n)$:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
$f(n)$	1	2	2	4	2	4	6	0	4	4	4	4	12	0	4	4	8	4	8	0	8	4	4	8	24	0	4	4	8	4

Kalle Saari (personal communication) asked about the growth rate of $f(n)$. The following results characterizes it.

Theorem 3. *We have $f(n) \leq n$ for $n \geq 4$. Furthermore, $f(n) = n$ infinitely often. Thus,* $\limsup_{n\geq 1} f(n)/n = 1$.

Proof. We start by verifying the following relations:

$$f(4n) = 2f(2n), \qquad (n \geq 2) \tag{3}$$
$$f(4n+1) = f(2n+1), \qquad (n \geq 0) \tag{4}$$
$$f(8n+2) = f(2n+1) + f(4n+3), \qquad (n \geq 1) \tag{5}$$
$$f(8n+3) = -f(2n+1) + f(4n+2) \qquad (n \geq 2) \tag{6}$$
$$f(8n+6) = -f(2n+1) + f(4n+2) + f(4n+3) \qquad (n \geq 2) \tag{7}$$
$$f(8n+7) = 2f(2n+1) + f(4n+3) \qquad (n \geq 3) \tag{8}$$

These can be verified in exactly the same way that we verified the system (2) earlier.

We now verify, by induction on n, that $f(n) \leq n$ for $n \geq 4$. The base case is $n = 4$, and $f(4) = 2$. Now assume $n \geq 5$. Otherwise,

- If $n \equiv 0 \pmod 4$, say $n = 4m$ and $m \geq 2$. Then $f(4m) = 2f(2m) \leq 2 \cdot 2m \leq 4m$ by (3) and induction.
- If $n \equiv 1 \pmod 4$, say $n = 4m+1$ for $m \geq 1$, then $f(4m+1) = f(2m+1)$ by (4). But $f(2m+1) \leq 2m+1$ by induction for $m \geq 2$. The case $m = 1$ corresponds to $f(5) = 4 \leq 5$.
- If $n \equiv 2 \pmod 8$, say $n = 8m+2$, then for $m \geq 2$ we have $f(8m+2) = f(2m+1) + f(4m+3) \leq 6m+4$ by induction, which is less than $8m+2$. If $m = 1$, then $f(10) = 4 < 10$.
- If $n \equiv 3 \pmod 8$, say $n = 8m+3$ for $m \geq 1$, then $f(8m+3) = -f(2m+1) + f(4m+2) \leq f(4m+2) \leq 4m+2$ by induction.
- If $n \equiv 6 \pmod 8$, say $n = 8m+6$, then $f(8m+6) = -f(2m+1) + f(4m+2) + f(4m+3) \leq f(4m+2) + f(4m+3) \leq 8m+5$ by induction, provided $m \geq 2$. For $m = 0$ we have $f(6) = 6$ and for $m = 1$ we have $f(14) = 4$.

– If $n \equiv 7 \pmod 8$, say $n = 8m+7$, then $f(8m+7) = 2f(2m+1)+f(4m+3) \leq 2(2m+1)+4m+3 = 8m+5$ for $m \geq 3$, by induction. The cases $m = 0, 1, 2$ can be verified by inspection.

This completes the proof that $f(n) \leq n$.

It remains to see that $f(n) = n$ infinitely often. We do this by showing that $f(n) = n$ for n of the form $3 \cdot 2^i$, $i \geq 1$. Let us prove this by induction on i. It is true for $i = 1$ since $f(6) = 6$. Otherwise $i \geq 2$, and using (3) we have $f(3 \cdot 2^{i+1}) = 2f(3 \cdot 2^i) = 2 \cdot 3 \cdot 2^i = 3 \cdot 2^{i+1}$ by induction. This also implies the claim $\limsup_{n \geq 1} f(n)/n = 1$.

5 Unbordered Factors of other Sequences

We can carry out similar computations for other famous sequences. In some cases the automata and the corresponding matrices are very large, which renders the computations time-consuming and the asymptotic behavior less transparent. We report on some of these computations, omitting the details.

Theorem 4. *Let* $\mathbf{r} = r_0r_1r_2\cdots = 00010010\cdots$ *denote the Rudin-Shapiro sequence, defined by* $r_n =$ *the number of occurrences, taken modulo 2, of '11' in the binary expansion of* n. *Let* $f_{\mathbf{r}}(n)$ *denote the number of unbordered factors of length* n *in* $\mathbf{r}$. *Then* $f_{\mathbf{r}}(n) \leq \frac{21}{8}n$ *for all* $n \geq 1$. *Furthermore if* $n = 2^i + 1$, *then* $f(n) = 21 \cdot 2^{i-3}$ *for* $i \geq 4$.

Theorem 5. *Let* $\mathbf{p} = p_0p_1p_2\cdots = 0100\cdots$ *be the so-called "period-doubling" sequence, defined by*

$$p_n = \begin{cases} 1, & \text{if } t_n = t_{n+1}; \\ 0, & \text{otherwise,} \end{cases}$$

where $t_0t_1t_2\cdots$ *is the Thue-Morse word* $\mathbf{t}$. *Note that* $\mathbf{p}$ *is the fixed point of the morphism* $0 \to 01$ *and* $1 \to 00$. *Then* $f_{\mathbf{p}}(n)$, *the number of unbordered factors of* $\mathbf{p}$ *of length* n, *is equal to* 2 *for all* $n \geq 1$.

The period-doubling sequence can be generalized to base $k \geq 2$, as follows:

$$\mathbf{p}_k := (\nu_k(n+1) \bmod 2)_{n \geq 0},$$

where $\nu_k(x)$ is the exponent of the largest power of k dividing x. For each k, the corresponding sequence $\mathbf{p}_k$ is a binary sequence that is k-automatic:

Theorem 6. *Let* k *be an integer* ≥ 2. *The sequence* $\mathbf{p}_k$ *is the fixed point of the morphism* φ_k, *where*

$$\varphi_k(0) = 0^{k-1}1$$
$$\varphi_k(1) = 0^k.$$

Proof. Note that $\mathbf{p}_k(n) = c$ iff $\nu_k(n+1) = 2j + c$ for some integer $j \geq 0$, and $c \in \{0,1\}$.

If $0 \leq a < k-1$, then $\mathbf{p}_k(kn+a) = \nu_k(kn+a+1) \bmod 2 = 0$. If $a = k-1$ we have $\mathbf{p}_k(kn+a) = \nu_k(kn+k) \bmod 2 = \nu_k(k(n+1)) \bmod 2 = (2j+c+1) \bmod 2$. Hence if $\mathbf{p}_k(n) = 0$, then $\mathbf{p}_k[kn..kn+k-1] = 0^{k-1}\,1$, while if $\mathbf{p}_k(n) = 1$, then $\mathbf{p}_k[kn..kn+k-1] = 0^k$. It follows that $\mathbf{p}_k$ is the fixed point of φ_k.

The generalized sequence $\mathbf{p}_k$ has the same property of unbordered factors as the period-doubling sequence:

Theorem 7. *The number of unbordered factors of* $\mathbf{p}_k$ *of length* n*, for* $k \geq 2$ *and* $n \geq 1$*, is equal to* 2*, and the two unbordered factors are reversals of each other.*

We begin with some useful lemmas.

Lemma 8. *Let* $x \in \{0,1\}^*$ *be a word. Then* $0^{k-1}\,\varphi_k(x)^R = \varphi_k(x^R)\,0^{k-1}$.

Proof. Suppose $x = a_1a_2\cdots a_n$, where each $a_i \in \{0,1\}$. If $a \in \{0,1\}$, let $\overline{a}$ denote $1-a$. Then

$$\begin{aligned}
0^{k-1}\,\varphi_k(x)^R &= 0^{k-1}\left(\prod_{1\leq i\leq n}\varphi_k(a_i)\right)^R \\
&= 0^{k-1}\left(\prod_{1\leq i\leq n}0^{k-1}\,\overline{a_i}\right)^R \\
&= 0^{k-1}\left(\prod_{1\leq i\leq n}\overline{a_{n+1-i}}\,0^{k-1}\right) \\
&= \left(\prod_{1\leq i\leq n}0^{k-1}\,\overline{a_{n+1-i}}\right)0^{k-1} \\
&= \left(\prod_{1\leq i\leq n}\varphi_k(a_{n+1-i})\right)0^{k-1} \\
&= \varphi_k(x^R)\,0^{k-1}.
\end{aligned}$$

Lemma 9. *If the word* w *is bordered, then* $\varphi_k(w)$ *is bordered.*

Proof. If w is bordered, then $w = xyx$ for $x \neq \epsilon$. Then $\varphi_k(w) = \varphi_k(x)\varphi_k(y)\varphi_k(x)$ is bordered.

Lemma 10. *If* w *is a factor of* $\mathbf{p}_k$*, then so is* w^R.

Proof. If w is a factor of $\mathbf{p}_k$, then it is a factor of some prefix $\mathbf{p}_k[0..k^i-1]$ for some $i \geq 1$. So it suffices to show that $\mathbf{p}_k[0..k^i-1]^R$ appears as a factor of $\mathbf{p}_k$. In fact, we claim that

$$\mathbf{p}_k[0..k^i-1]^R = \mathbf{p}_k[k^i-1..2k^i-2].$$

To see this, it suffices to observe that $\nu_k(k^i-a) = \nu_k(k^i+a)$ for $0 \leq a < k^i$.

The following lemma describes the unbordered factors of φ_k. If $w = 0^a x$, then by $0^{-a}\, w$ we mean the word x.

Lemma 11. *(a) If w is an unbordered factor of $\mathbf{p}_k$ and $|w| \equiv 0 \pmod{k}$, then $w = \varphi_k(x)$ or $w = \varphi_k(x)^R$, for some unbordered factor x of $\mathbf{p}_k$ with $|x| = |w|/k$.*

(b) If w is an unbordered factor of $\mathbf{p}_k$ and $|w| \equiv a \pmod{k}$ for $0 < a < k$, then $w = 0^{a-k}\, \varphi_k(x)$ or $w = \varphi_k(x)^R\, 0^{a-k}$, for some unbordered factor x of $\mathbf{p}_k$ with $|x| = (|w| - a)/k + 1$.

Proof. (a): Suppose that $w = \mathbf{p}_k[i..i + kn - 1]$ for some integer i. There are two cases to consider: $\mathbf{p}_k[i] = 0$ and $\mathbf{p}_k[i] = 1$.

Suppose $\mathbf{p}_k[i] = 0$. Since $\mathbf{w}$ is unbordered, we have $\mathbf{p}_k[i + kn - 1] = 1$. Then $\nu_k(i + kn) \geq 1$, so $i + kn = km$ for some $m \geq 0$. Then $i = k(m - n)$ is a multiple of k, so $w = \varphi_k(x)$, where $x = \mathbf{p}_k[i/k..i/k + n - 1]$. Note that $|x| = |w|/k$. Finally, Lemma 9 shows that x is unbordered.

Suppose $\mathbf{p}_k[i] = 1$. Since w is unbordered, we have $\mathbf{p}_k[i + kn - 1] = 0$. From Lemma 10 we know that w^R is also a factor of $\mathbf{p}_k$ (and also is unbordered). Then from the previous paragraph, we see that $w^R = \varphi_k(x)$ for some unbordered factor x of $\mathbf{p}_k$, with $|x| = |w|/k$. Then $w = \varphi_k(x)^R$, as desired.

(b): Suppose that $w = \mathbf{p}_k[i..i + kn + a - 1]$ for $0 < a < k$. There are two cases to consider: $\mathbf{p}_k[i] = 0$ and $\mathbf{p}_k[i] = 1$.

Suppose that $\mathbf{p}_k[i] = 0$. Since w is unbordered, we know that $\mathbf{p}_k[i+kn+a-1] = 1$. Then $\nu_k(i + kn + a) \geq 1$, so $i + kn + a = km$ for some $m \geq 0$. Then $i - (k - a) = k(m - n - 1)$ is a multiple of k. Hence

$$0^{k-a}\, w = \mathbf{p}_k[i - (k - a)..i + kn + a - 1] = \varphi_k(\mathbf{p}_k[(i + a)/k - 1..(i + a)/k + n - 1]).$$

Let $x = \mathbf{p}_k[(i + a)/k - 1..(i + a)/k + n - 1]$. Then $w = 0^{a-k}\, \varphi_k(x)$, and $|x| = (|w| - a)/k + 1$. If x is bordered, then using Lemma 9 we have that $0^{k-a}\, w$ has a border of length $\geq k$, so w has a border of length at least a, a contradiction.

Suppose that $\mathbf{p}_k[i] = 1$. Since w is unbordered, we know that $\mathbf{p}_k[i+kn+a-1] = 0$. Then by Lemma 10 we know that w^R is also an unbordered factor of $\mathbf{p}_k$. Then from the previous paragraph, we get that $w^R = 0^{a-k}\, \varphi_k(x)$ for some unbordered factor x of $\mathbf{p}_k$ where $|x| = (|w| - a)/k + 1$. So $w = \varphi_k(x)^R\, 0^{a-k}$, as desired.

Lemma 12. *Let x be a word and $w = 1x0$ be an unbordered word. Then $0^i \varphi_k(x0)$ is unbordered for $1 \leq i \leq k$.*

Proof. If $i = k$ then $0^k \varphi_k(x0) = \varphi_k(1x0) = \varphi_k(w)$. Suppose $\varphi_k(w)$ is bordered; then there exist $u \neq \epsilon$ and v such that $\varphi_k(w) = uvu$. Since $\varphi_k(0) = 0^{k-1}1$, we know u ends in 1. But since u is a prefix of $\varphi_k(w)$ that ends in 1, it follows that $|u| \equiv 0 \pmod{k}$, and so u is the image of some word r under φ_k. Hence w begins and ends with r, a contradiction.

Now assume $1 \leq i < k$ and $0^i \varphi_k(x0)$ is bordered. Then there exist $u \neq \epsilon$ and v such that $0^i \varphi_k(x0) = uvu$; note that u must end in 1. It follows that

$$\varphi_k(w) = \varphi_k(1x0) = 0^k \varphi_k(x0) = 0^{k-i}(0^i \varphi_k(x0)) = 0^{k-i} uvu.$$

Since $0^{k-i}u$ and $0^{k-i}uvu$ both end in 1 and $0^{k-i}uvu = \varphi_k(w)$, we have $|vu| \equiv 0 \pmod{k}$. Hence $|u| \equiv i \pmod{k}$. It follows that $0^{k-i}uv$ ends in 0^k, so $0^{k-i}uvu = \varphi_k(w)$ begins and ends in $0^{k-i}u$, a contradiction.

We are now ready for the proof of Theorem 7.

Proof. First, we show that there is at least one unbordered factor of every length, by induction on n. The base cases are $n < 2k$, and are left to the reader. Otherwise $n \geq 2k$. Write $n = kn' + i$ where $1 \leq i \leq k$. By induction there is an unbordered word w of length $n' + 1$. Using Lemma 10, we can assume that w begins with 1 and ends with 0, say $w = 1x0$. By Lemma 12 we have that $0^i \varphi_k(x0)$ is unbordered, and it is of length $i + kn' = n$.

It remains to prove there are exactly 2 unbordered factors of every length.

If $n \leq 2k$, then it is easy to see that the only unbordered factors are $1\,0^{n-1}$ and $0^{n-1}\,1$.

Now assume $n > 2k$ and that there are only two unbordered factors of length n' for all $n' < n$; we prove it for n. Let w be an unbordered factor of length n.

If $n \equiv 0 \pmod{k}$, then by Lemma 11 (a), we know that either $w = \phi_k(x)$ or $w = \phi_k(x)^R$, where x is an unbordered factor of length n/k. By induction there are exactly 2 unbordered factors of length n/k; by Lemma 10 they are reverses of each other. Let x be such an unbordered factor; since $|x| = n/k > 2$, either x begins with 0 and ends with 1, or begins with 1 and ends with 0. In the former case, the image $w = \varphi_k(x)$ begins and ends with 0, a contradiction. So x begins with 1 and ends with 0. But there is only one such factor, so there are only two possibilities for w.

Otherwise let $a = n \bmod k$; then $0 < a < k$. By Lemma 11 (b), we know that $w = 0^{a-k}\,\varphi_k(x)$ or $w = \varphi_k(x)^R\,0^{a-k}$, where x is an unbordered factor of length $(|w| - a)/k + 1 \geq 2$. By induction there are exactly 2 such unbordered words; by Lemma 10 they are reverses of each other. Let x be such an unbordered factor; then either x begins with 0 and ends with 1, or begins with 1 and ends with 0. Let us call them x_0 and x_1, respectively, with $x_0 = x_1^R$. Now $\varphi_k(x_0)$ begins with $0^{k-1}\,1$, and ends with 0^k. Hence, provided $a \neq 1$, we see that $w = 0^{a-k}\,\varphi_k(x_0)$ begins with 0 and ends with 0, a contradiction. If $a = 1$, Lemma 11 (b) gives the two factors $0^{1-k}\,\varphi_k(x_0)$ and $\varphi_k(x_0)^R\,0^{1-k}$. The former begins with 1 and ends with 0; the latter begins with 0 and ends with 1.

In the latter case, x_1 begins with 1 and ends with 0. There is only one such x_1 (by induction), and then either $w = 0^{a-k}\,\varphi_k(x_1)$ or $w = \varphi_k(x_1)^R\,0^{a-k}$, giving at most two possibilities for w. In the case $a = 1$, these two factors would seem to give a total of four factors of length n. However, there are only two, since

$$0^{1-k}\,\varphi_k(x_0) = 0^{1-k}\,\varphi_k(x_1^R) = \varphi_k(x_1)^R\,0^{1-k}$$
$$\varphi_k(x_0)^R\,0^{1-k} = 0^{1-k}\,\varphi_k(x_0^R) = 0^{1-k}\,\varphi_k(x_1)$$

This completes the proof.

References

1. Allouche, J.-P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press (2003)
2. Charlier, É., Rampersad, N., Shallit, J.: Enumeration and Decidable Properties of Automatic Sequences. In: Mauri, G., Leporati, A. (eds.) DLT 2011. LNCS, vol. 6795, pp. 165–179. Springer, Heidelberg (2011)
3. Cobham, A.: Uniform tag sequences. Math. Systems Theory 6, 164–192 (1972)
4. Currie, J.D., Saari, K.: Least periods of factors of infinite words. RAIRO Inform. Théor. App. 43(1), 165–178 (2009)
5. Duval, J.-P., Harju, T., Nowotka, D.: Unbordered factors and Lyndon words. Discrete Math. 308, 2261–2264 (2008)
6. Ehrenfeucht, A., Silberger, D.M.: Periodicity and unbordered segments of words. Discrete Math. 26, 101–109 (1979)
7. Goč, D., Henshall, D., Shallit, J.: Automatic Theorem-Proving in Combinatorics on Words. In: Moreira, N., Reis, R. (eds.) CIAA 2012. LNCS, vol. 7381, pp. 180–191. Springer, Heidelberg (2012)
8. Harju, T., Nowotka, D.: Periodicity and unbordered words: a proof of the extended Duval conjecture. J. Assoc. Comput. Mach. 54, Article 20 (2007)
9. Holub, S.: A proof of the extended Duval's conjecture. Theoret. Comput. Sci. 339, 61–67 (2005)
10. Holub, S., Nowotka, D.: On the relation between periodicity and unbordered factors of finite words. Internat. J. Found. Comp. Sci. 21, 633–645 (2010)
11. Nielsen, P.T.: A note on bifix-free sequences. IEEE Trans. Inform. Theory IT-19, 704–706 (1973)
12. Silberger, D.M.: Borders and roots of a word. Portugal. Math. 30, 191–199 (1971)
13. Silberger, D.M.: How many unbordered words? Ann. Soc. Math. Polon. Ser. I: Comment. Math. 22, 143–145 (1980)

Primitive Words and Lyndon Words in Automatic and Linearly Recurrent Sequences

Daniel Goč[1], Kalle Saari[2], and Jeffrey Shallit[1]

[1] School of Computer Science
University of Waterloo, Waterloo, ON N2L 3G1, Canada
{dgoc,shallit}@cs.uwaterloo.ca
[2] Mathematics and Statistics, University of Winnipeg
515 Portage Avenue, Winnipeg, MB R3B 2E9, Canada
kasaar2@gmail.com

Abstract. We investigate questions related to the presence of primitive words and Lyndon words in automatic and linearly recurrent sequences. We show that the Lyndon factorization of a k-automatic sequence is itself k-automatic. We also show that the function counting the number of primitive factors (resp., Lyndon factors) of length n in a k-automatic sequence is k-regular. Finally, we show that the number of Lyndon factors of a linearly recurrent sequence is bounded.

Keywords: Lyndon word, Lyndon factorization, primitive word, automatic sequence, linearly recurrent sequence.

1 Introduction

We start with some basic definitions. A nonempty word w is called a *power* if it can be written in the form $w = x^k$, for some integer $k \geq 2$. Otherwise w is called *primitive*. Thus `murmur` is a power, but `murder` is primitive. A word y is a *factor* of a word w if there exist words x, z such that $w = xyz$. If further $x = \epsilon$ (resp., $z = \epsilon$), then y is a *prefix* (resp., *suffix*) of w. A prefix or suffix of a word w is called *proper* if it is unequal to w.

Let Σ be an ordered alphabet. We recall the usual definition of lexicographic order on the words in Σ^*. We write $w < x$ if either

(a) w is a proper prefix of x; or
(b) there exist words y, z, z' and letters $a < b$ such that $w = yaz$ and $x = ybz'$.

For example, using the usual ordering of the alphabet, we have `common` $<$ `con` $<$ `conjugate`. As usual, we write $w \leq x$ if $w < x$ or $w = x$.

A word w is a *conjugate* of a word x if there exist words u, v such that $w = uv$ and $w = vu$. Thus, for example, `enlist` and `listen` are conjugates. A word is said to be *Lyndon* if it is primitive and lexicographically least among all its conjugates. Thus, for example, `academy` is Lyndon, while `googol` and `googoo` are not. Lyndon words have received a great deal of attention in the combinatorics on words literature. For example, a finite word is Lyndon if and only if it is lexicographically less than each of its proper suffixes [9] and this can be tested in linear time.

We now turn to (right-) infinite words. We write an infinite word in boldface, as $\mathbf{x} = a_0 a_1 a_2 \cdots$ and use indexing starting at 0. For $i \leq j+1$, we let $[i..j]$ denote the set $\{i, i+1, \ldots, j\}$. (If $i = j+1$ we get the empty set.) We let $\mathbf{x}[i..j]$ denote the word $a_i a_{i+1} \cdots a_j$. Similarly, $[i..\infty]$ denotes the infinite set $\{i, i+1, \ldots\}$ and $\mathbf{x}[i..\infty]$ denotes the infinite word $a_i a_{i+1} \cdots$.

An infinite word or sequence $\mathbf{x} = a_0 a_1 a_2 \cdots$ is said to be *k-automatic* if there is a deterministic finite automaton (with outputs associated with the states) that, on input n expressed in base k, reaches a state q with output $\tau(q)$ equal to a_n. For more details, see [6] or [2]. In several previous papers [1,5,14,16,10], we have developed a technique to show that many properties of automatic sequences are decidable. The fundamental tool is the following:

Theorem 1. *Let $P(n)$ be a predicate associated with a k-automatic sequence $\mathbf{x}$, expressible using addition, subtraction, comparisons, logical operations, indexing into $\mathbf{x}$, and existential and universal quantifiers. Then there is a computable finite automaton accepting the base-k representations of those n for which $P(n)$ holds. Furthermore, we can decide if $P(n)$ holds for at least one n, or for all n, or for infinitely many n.*

If a predicate is constructed as in the previous theorem, we just say it is "expressible". Any expressible predicate is decidable. As an example, we prove

Theorem 2. *Let $\mathbf{x}$ be a k-automatic sequence. The predicate $P(i,j)$ defined by "$\mathbf{x}[i..j]$ is primitive" is expressible.*

Proof. (due to Luke Schaeffer) It is easy to see that a word is a power if and only if it is equal to some cyclic shift of itself, other than the trivial shift. Thus a word is a power if and only if there is a d, $0 < d < j - i + 1$, such that $x[i..j-d] = x[i+d..j]$ and $x[j-d+1..j] = x[i..i+d-1]$. A word is primitive if there is no such d.

Theorem 3. *Let $\mathbf{x}$ be a k-automatic sequence. The predicate $LL(i,j,m,n)$ defined by "$\mathbf{x}[i..j] < \mathbf{x}[m..n]$" is expressible.*

Proof. We have $\mathbf{x}[i..j] < \mathbf{x}[m..n]$ if and only if either

(a) $j - i < n - m$ and $\mathbf{x}[i..j] = \mathbf{x}[m..m+j-i]$; or
(b) there exists $t < \min(j-i, n-m)$ such that $\mathbf{x}[i..i+t] = \mathbf{x}[m..m+t]$ and $\mathbf{x}[i+t+1] < \mathbf{x}[m+t+1]$.

Theorem 4. *Let $\mathbf{x}$ be a k-automatic sequence. The predicate $L(i,j)$ defined by "$\mathbf{x}[i..j]$ is a Lyndon word" is expressible.*

Proof. It suffices to check that $\mathbf{x}[i..j]$ is lexicographically less than each of its proper suffixes, that is, that $LL(i,j,i',j)$ holds for all i' with $i < i' \leq j$.

We can extend the definition of lexicographic order to infinite words in the obvious way. We can extend the definition of Lyndon words to (right-) infinite words as follows: an infinite word $\mathbf{x} = a_0 a_1 a_2 \cdots$ is Lyndon if it is lexicographically less than all its suffixes $\mathbf{x}[j..\infty] = a_j a_{j+1} \cdots$ for $j \geq 1$. Then we have the following theorems.

Theorem 5. *Let* $\mathbf{x}$ *be a* k*-automatic sequence. The predicate* $LL_\infty(i, j)$ *defined by "*$\mathbf{x}[i..\infty] < \mathbf{x}[j..\infty]$*" is expressible.*

Proof. This is equivalent to $\exists t \geq 0$ such that $\mathbf{x}[i..i+t-1] = \mathbf{x}[j..j+t-1]$ and $\mathbf{x}[i+t] < \mathbf{x}[j+t]$.

Theorem 6. *Let* $\mathbf{x}$ *be a* k*-automatic sequence. The predicate* $L_\infty(i)$ *defined by "*$\mathbf{x}[i..\infty]$ *is an infinite Lyndon word" is expressible.*

Proof. This is equivalent to $LL_\infty(i, j)$ holding for all $j > i$.

2 Lyndon Factorization

Siromoney et al. [17] proved that every infinite word $\mathbf{x} = a_0 a_1 a_2 \cdots$ can be factorized uniquely in exactly one of the following two ways:

(a) as $\mathbf{x} = w_1 w_2 w_3 \cdots$ where each w_i is a finite Lyndon word and $w_1 \geq w_2 \geq w_3 \cdots$; or
(b) as $\mathbf{x} = w_1 w_2 w_3 \cdots w_r \mathbf{w}$ where w_i is a finite Lyndon word for $1 \leq i \leq r$, and $\mathbf{w}$ is an infinite Lyndon word, and $w_1 \geq w_2 \geq \cdots \geq w_r \geq \mathbf{w}$.

If (a) holds we say that the Lyndon factorization of $\mathbf{x}$ is infinite; otherwise we say it is finite.

Ido and Melançon [13,12] gave an explicit description of the Lyndon factorization of the Thue-Morse word $\mathbf{t}$ and the period-doubling sequence (among other things). (Recall that the Thue-Morse word is given by $\mathbf{t}[n]$ = the number of 1's in the binary expansion of n, taken modulo 2.) For the Thue-Morse word, this factorization is given by

$$\mathbf{t} = w_1 w_2 w_3 w_4 \cdots = (011)(01)(0011)(00101101) \cdots,$$

where each term in the factorization, after the first, is double the length of the previous. Séébold [15] and Černý [4] generalized these results to other related automatic sequences.

In this section, generalizing the work of Ido, Melançon, Séébold, and Černý, we prove that the Lyndon factorization of a k-automatic sequence is itself k-automatic. Of course, we need to explain how the factorization is encoded. The easiest and most natural way to do this is to use an infinite word over $\{0, 1\}$, where the 1's indicate the positions where a new term in the factorization begins. Thus the i'th 1, for $i \geq 0$, appears at index $|w_1 w_2 \cdots w_i|$. For example, for the Thue-Morse word, this encoding is given by

$$100101000100000001 \cdots.$$

If the factorization is infinite, then there are infinitely many 1's in its encoding; otherwise there are finitely many 1's.

In order to prove the theorem, we need a number of results. We draw a distinction between a *factor* f of $\mathbf{x}$ (which is just a word) and an *occurrence* of that factor (which specifies the exact position at which f occurs). For example, in the Thue-Morse word $\mathbf{t}$, the factor 0110 occurs as $\mathbf{x}[0..3]$ and $\mathbf{x}[11..15]$ and many other places. We call $[0..3]$ and $[11..15]$, and so forth, the *occurrences* of 0110. An occurrence is said to be Lyndon if the word at that position is Lyndon. We say an occurrence $O_1 = [i..j]$ is *inside* an occurrence $O_2 = [i'..j']$ if $i' \leq i$ and $j' \geq j$. If, in addition, either $i' < i$ or $j < j'$ (or both), then we say O_1 is *strictly inside* O_2. These definitions are easily extended to the case where j or j' are equal to ∞, and they correspond to the predicates I (inside) and SI (strictly inside) given below:

$$\begin{aligned} I(i,j,i',j') \text{ is } \quad & i' \leq i \text{ and } j' \geq j \\ SI(i,j,i',j') \text{ is } \quad & I(i,j,i',j') \text{ and } ((i' < i) \text{ or } (j' > j)) \end{aligned}$$

An infinite Lyndon factorization

$$\mathbf{x} = w_1 w_2 w_3 \cdots$$

then corresponds to an infinite sequence of occurrences

$$[i_1..j_1], [i_2..j_2], \cdots$$

where $w_n = \mathbf{x}[i_n..j_n]$ and $i_{n+1} = j_n + 1$ for $n \geq 1$, while a finite Lyndon factorization

$$\mathbf{x} = w_1 w_2 \cdots w_r \mathbf{w}$$

corresponds to a finite sequence of occurrences

$$[i_1..j_1], [i_2..j_2], \ldots, [i_r..j_r], [i_{r+1}..\infty]$$

where $w_n = \mathbf{x}[i_n..j_n]$ and $i_{n+1} = j_n + 1$ for $1 \leq n \leq r$.

Theorem 7. *Let* $\mathbf{x}$ *be an infinite word. Every Lyndon occurrence in* $\mathbf{x}$ *appears inside a term of the Lyndon factorization of* $\mathbf{x}$.

Proof. We prove the result for infinite Lyndon factorizations; the result for finite factorizations is exactly analogous.

Suppose the factorization is $\mathbf{x} = w_1 w_2 w_3 \cdots$. It suffices to show that no Lyndon occurrence can span the boundary between two terms of the factorization. Suppose, contrary to what we want to prove, that $u w_i w_{i+1} \cdots w_j v$ is a Lyndon word for some u that is a nonempty suffix of w_{i-1} (possibly equal to w_{i-1}), and v that is a nonempty prefix of w_{j+1} (possibly equal to w_{j+1}), and $i \leq j+1$. (If $i = j+1$ then there are no w_i's at all between u and v.)

Since u is a suffix of w_{i-1} and w_{i-1} is Lyndon, we have $u \geq w_{i-1}$. On the other hand, by the Lyndon factorization definition we have $w_{i-1} \geq w_i \geq \cdots \geq$

$w_j \geq w_{j+1}$. But v is a prefix of w_{j+1}, so just by the definition of lexicographic ordering we have $w_{j+1} \geq v$. Putting this all together we get $u \geq v$. So $ux \geq v$ for all words x.

On the other hand, since $uw_i \cdots w_j v$ is Lyndon, it must be lexicographically less than any proper suffix — for instance, v. So $uw_i \cdots w_j v < v$. Take $x = w_i \cdots w_j v$ to get a contradiction with the conclusion in the previous paragraph.

Corollary 8. *The occurrence $[i..j]$ corresponds to a term in the Lyndon factorization of* $\mathbf{x}$ *if and only if*

(a) $[i..j]$ *is Lyndon; and*
(b) $[i..j]$ *does not occur strictly inside any other Lyndon occurrence.*

Proof. Suppose $[i..j]$ corresponds to a term w_n in the Lyndon factorization of $\mathbf{x}$. Then evidently $[i..j]$ is Lyndon. If it occurred strictly inside some other Lyndon occurrence, say $[i'..j']$, then we know from Theorem 7 that $[i'..j']$ itself lies in inside some $[i_m, j_m]$, so $[i..j]$ must lie strictly inside $[i_m, j_m]$, which is clearly impossible.

Now suppose $[i..j]$ is Lyndon and does not occur strictly inside any other Lyndon occurrence. From Theorem 7 $[i..j]$ must occur inside some term of the factorization $[i'..j']$. If $[i..j] \neq [i'..j']$ then $[i..j]$ lies strictly inside $[i'..j']$, a contradiction. So $[i..j] = [i'..j']$ and hence corresponds to a term of the factorization.

Corollary 9. *The predicate $LF(i,j)$ defined by "$[i..j]$ corresponds to a term of the Lyndon factorization of* $\mathbf{x}$*" is expressible.*

Proof. Indeed, by Corollary 8, the predicate $LF(i,j)$ can be defined by

$$L(i,j) \text{ and } \forall\ i', j'\ (SI(i,j,i',j') \implies \neg L(i',j')).$$

We can now prove the main result of this section.

Theorem 10. *Using the encoding mentioned above, the Lyndon factorization of a k-automatic sequence is itself k-automatic.*

Proof. Using the technique of [1], we can create an automaton that on input i expressed in base k, guesses j and checks if $LF(i,j)$ holds. If so, it outputs 1 and otherwise 0. To get the last i in the case that the Lyndon factorization is finite, we also accept i if $L_\infty(i)$ holds.

We also have

Theorem 11. *Let* $\mathbf{x}$ *be a k-automatic sequence. It is decidable if the Lyndon factorization of* $\mathbf{x}$ *is finite or infinite.*

Proof. The construction given above in the proof of Theorem 10 produces an automaton that accepts finitely many distinct i (expressed in base k) if and only if the Lyndon factorization of $\mathbf{x}$ is finite.

We programmed up our method and found the Lyndon factorization of the Thue-Morse sequence $\mathbf{t}$, the period-doubling sequence $\mathbf{d}$, the paperfolding sequence $\mathbf{p}$, and the Rudin-Shapiro sequence $\mathbf{r}$, and their negations. (The results for Thue-Morse and the period-doubling sequence were already given in [12], albeit in a different form.) Recall that the period-doubling sequence is defined by $\mathbf{p}[n] = |\mathbf{t}[n+1] - \mathbf{t}[n]|$. The paperfolding sequence $\mathbf{p} = 0010011 \cdots$ arises from the limit of the sequence (f_n), where $f_0 = 0$ and $f_{n+1} = f_n 0 \overline{f_n}^R$, where R denotes reversal and $\overline{x}$ maps 0 to 1 and 1 to 0. Finally, the Rudin-Shapiro sequence $\mathbf{r}$ is defined by $\mathbf{r}[n] =$ the number of (possibly overlapping) occurrences of 11 in the binary expansion of n, taken modulo 2. The results are given in the theorem below.

Theorem 12. *The occurrences corresponding to the Lyndon factorization of each word is as follows:*

– *the Thue-Morse sequence* $\mathbf{t}$*:* $[0..2], [3..4], [5..8], [9..16], \ldots, [2^i + 1..2^{i+1}], \ldots$*;*
– *the negated Thue-Morse sequence* $\overline{\mathbf{t}}$*:* $[0..0], [1..\infty]$*;*
– *the Rudin-Shapiro sequence* $\mathbf{r}$*:* $[0..6], [7..14], [15..30], \ldots, [2^i - 1..2^{i+1} - 2], \ldots$*;*
– *the negated Rudin-Shapiro sequence*
$\overline{\mathbf{r}}$*:* $[0..0], [1..1], [2..2], [3..10], [11..42], [43..46], \ldots, [4^i - 4^{i-1} - 4^{i-2} - 1..4^i - 4^{i-1} - 2], [4^i - 4^{i-1} - 1..4^{i+1} - 4^i - 4^{i-1} - 1], \ldots$*;*
– *the paperfolding sequence* $\mathbf{p}$*:* $[0..6], [7..14], [15..30], \ldots, [2^i - 1..2^{i+1} - 2], \ldots$*;*
– *the negated paperfolding sequence* $\overline{\mathbf{p}}$*:* $[0..0], [1..1], [2..4], [5..9], [10..20], [21..84], \ldots, [(4^i - 1)/3..4(4^i - 1)/3], \ldots$*;*
– *the period-doubling sequence* $\mathbf{d}$*:* $[0..0], [1..4], [5..20], [21..84], \ldots,$
$[(4^i - 1)/3..4(4^i - 1)/3], \ldots$*;*
– *the negated period-doubling sequence* $\overline{\mathbf{d}}$*:* $[0..1], [2..9], [10..41], [42..169], \ldots,$
$[2(4^i - 1)/3..2(4^{i+1} - 1)/3 - 1], \ldots$*.*

3 Enumeration

There is a useful generalization of k-automatic sequences to sequences over $\mathbb{N}$, the non-negative integers. A sequence $(a_n)_{n \geq 0}$ over $\mathbb{N}$ is called k-regular if there exist vectors u and v and a matrix-valued morphism μ such that $a_n = u\mu(w)v$, where w is the base-k representation of n. For more details, see [3].

The subword complexity function $\rho(n)$ of an infinite sequence $\mathbf{x}$ counts the number of distinct length-n factors of $\mathbf{x}$. There are also many variations, such as counting the number of palindromic factors or unbordered factors. If $\mathbf{x}$ is k-automatic, then all three of these are k-regular sequences [1]. We now show that the same result holds for the number $\rho^P_{\mathbf{x}}(n)$ of primitive factors of length n and for the number $\rho^L_{\mathbf{x}}$ of Lyndon factors of length n. We refer to these two quantities as the "primitive complexity" and "Lyndon complexity", respectively.

Theorem 13. *The function counting the number of length-n primitive (resp., Lyndon) factors of a k-automatic sequence* $\mathbf{x}$ *is k-regular.*

Proof. By the results of [5], it suffices to show that there is an automaton accepting the base-k representations of pairs (n, i) such that the number of i's associated with each n equals the number of primitive (resp., Lyndon) factors of length n.

To do so, it suffices to show that the predicate $P(n, i)$ defined by "the factor of length n beginning at position i is primitive (resp., Lyndon) and is the first occurrence of that factor in $\mathbf{x}$" is expressible. This is just

$$P(i, i+n-1) \quad \text{and} \quad \forall j < i \; \mathbf{x}[i..i+n-1] \neq \mathbf{x}[j..j+n-1],$$

(resp.,

$$L(i, i+n-1) \quad \text{and} \quad \forall j < i \; \mathbf{x}[i..i+n-1] \neq \mathbf{x}[j..j+n-1]).$$

We used our method to compute these sequences for the Thue-Morse sequence, and the results are given below.

Theorem 14. *Let $\rho^L_{\mathbf{t}}(n)$ denote the number of Lyndon factors of length n of the Thue-Morse sequence. Then*

$$\rho^L_{\mathbf{t}}(n) = \begin{cases} 1, & \text{if } n = 2^k \text{ or } 5 \cdot 2^k \text{ for } k \geq 1\ ; \\ 2, & \text{if } n = 1 \text{ or } n = 5 \text{ or } n = 3 \cdot 2^k \text{ for } k \geq 0; \\ 0, & \text{otherwise.} \end{cases}$$

Theorem 15. *Let $\rho^P_{\mathbf{t}}(n)$ denote the number of primitive factors of length n of the Thue-Morse sequence. Then*

$$\rho^P_{\mathbf{t}}(n) = \begin{cases} 3 \cdot 2^t - 4, & \text{if } n = 2^t; \\ 4n - 2^t - 4, & \text{if } 2^t + 1 \leq n < 3 \cdot 2^{t-1}; \\ 5 \cdot 2^t - 6, & \text{if } n = 3 \cdot 2^{t-1}; \\ 2n + 2^{t+1} - 2, & \text{if } 3 \cdot 2^{t-1} < n < 2^{t+1}. \end{cases}$$

We can also state a similar result for the Rudin-Shapiro sequence.

Theorem 16. *Let $\rho^L_{\mathbf{r}}(n)$ denote the Lyndon complexity of the Rudin-Shapiro sequence. Then $\rho^L_{\mathbf{r}}(n) \leq 8$ for all n. This sequence is 2-automatic and there is an automaton of 2444 states that generates it.*

Proof. The proof was carried out by machine computation, and we briefly summarize how it was done.

First, we created an automaton A to accept all pairs of integers (n, i), represented in base 2, such that the factor of length n in $\mathbf{r}$, starting at position i, is a Lyndon factor, and is the first occurrence of that factor in $\mathbf{r}$. Thus, the number of distinct integers i associated with each n is $\rho^L_{\mathbf{r}}(n)$. The automaton A has 102 states.

Using the techniques in [5], we then used A to create matrices M_0 and M_1 of dimension 102×102, and vectors v, w such that $vM_x w = \rho^L_{\mathbf{r}}(n)$, if x is the base-2 representation of n. Here if $x = a_1 a_2 \cdots a_i$, then by M_x we mean the product $M_{a_1} M_{a_2} \cdots M_{a_i}$.

From this we then created a new automaton A' where the states are products of the form vM_x for binary strings x and the transitions are on 0 and 1. This automaton was built using a breadth-first approach, using a queue to hold states whose targets on 0 and 1 are not yet known. From Theorem 24 in the next section, we know that $\rho^L_{\mathbf{r}}(n)$ is bounded, so that this approach must terminate. It did so at 2444 states, and the product of the vM_x corresponding to each state with w gives an integer less than or equal to 8, thus proving the desired result and also providing an automaton to compute $\rho^L_{\mathbf{r}}(n)$.

Remark 17. Note that the Lyndon complexity functions in Theorems 14 and 16 are bounded. This will follow more generally from Theorem 24 below.

4 Finite Factorizations

Of course, the original Lyndon factorization was for finite words: every finite nonempty word x can be factored uniquely as a nonincreasing product $w_1 w_2 \cdots w_m$ of Lyndon words. We can apply this theorem to all prefixes of a k-automatic sequence. It is then natural to wonder if a *single* automaton can encode *all* the Lyndon factorizations of *all* finite prefixes. The answer is yes, as the following result shows.

Theorem 18. *Suppose* $\mathbf{x}$ *is a* k*-automatic sequence. Then there is an automaton* A *accepting*

$$\{(n,i)_k \ : \ \textit{the Lyndon factorization of}\ \mathbf{x}[0..n-1]\ \textit{is}\ w_1 w_2 \cdots w_m \\ \textit{with}\ w_m = \mathbf{x}[i..n-1]\}.$$

Proof. As is well-known [9], if $w_1 w_2 \cdots w_m$ is the Lyndon factorization of x, then w_m is the lexicographically least suffix of x. So to accept $(n,i)_k$ we find i such that $\mathbf{x}[i..n-1] < \mathbf{x}[j..n-1]$ for $0 \le j < n$ and $i \ne j$.

Given A, we can find the complete factorization of any prefix $\mathbf{x}[0..n-1]$ by using this automaton to find the appropriate i (as described in [11]) and then replacing n with i.

We carried out this construction for the Thue-Morse sequence, and the result is shown below in Figure 1.

In a similar manner, there is an automaton that encodes the factorization of *every* factor of a k-automatic sequence:

Theorem 19. *Suppose* $\mathbf{x}$ *is a* k*-automatic sequence. Then there is an automaton* A' *accepting*

$$\{(i,j,l)_k \ : \ \textit{the Lyndon factorization of}\ \mathbf{x}[i..j-1]\ \textit{is}\ w_1 w_2 \cdots w_m \\ \textit{with}\ w_m = \mathbf{x}[l..j-1]\}.$$

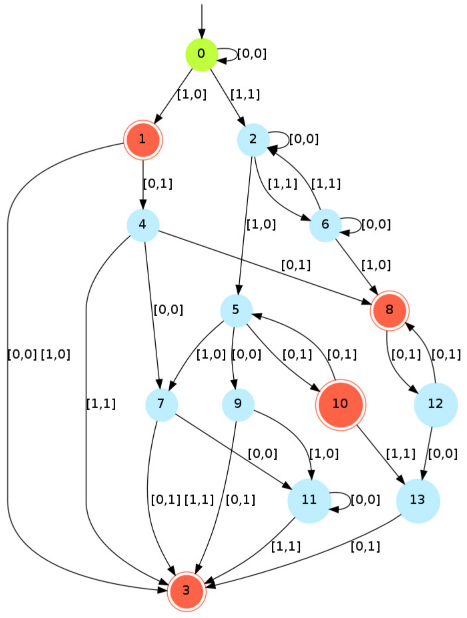


Fig. 1. A finite automaton accepting the base-2 representation of (n, i) such that the Lyndon factorization of $\mathbf{t}[0..n-1]$ ends in the term $\mathbf{t}[i..n-1]$

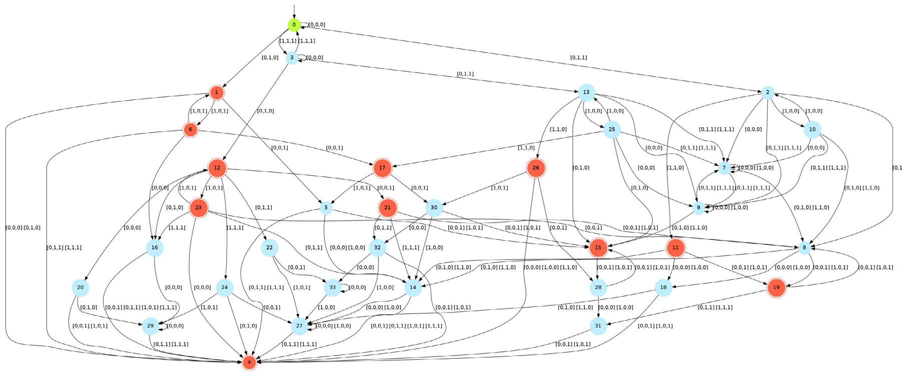


Fig. 2. A finite automaton accepting the base-2 representation of (i, j, l) such that the Lyndon factorization of $\mathbf{t}[i..j-1]$ ends in the term $\mathbf{t}[l..j-1]$

We calculated A' for the Thue-Morse sequence using our method. It is a 34-state machine and is displayed in Figure 2.

Another quantity of interest is the number of terms in the Lyndon factorization of each prefix.

Theorem 20. *Let x be a k-automatic sequence. Then the sequence $(f(n))_{n \geq 0}$ defined by*

$$f(n) = \textit{the number of terms in the Lyndon factorization of } \mathbf{x}[0..n]$$

is k-regular.

Proof. We construct an automaton to accept $\{(n, i) \; : \; \exists j \leq n$ such that $L(i, j)$ and if $SI(i, j, i', j')$ and $0 \leq i' \leq j' \leq n$ then $\neg L(i', j')\}$.

For the Thue-Morse sequence the corresponding sequence satisfies the relations

$$\begin{aligned}
f(4n+1) &= -f(2n) + f(2n+1) + f(4n) \\
f(8n+2) &= -f(2n) + f(4n) + f(4n+2) \\
f(8n+3) &= -f(2n) + f(4n) + f(4n+3) \\
f(8n+6) &= -f(2n) - f(4n+2) + 3f(4n+3) \\
f(8n+7) &= -f(2n) + 2f(4n+3) \\
f(16n) &= -f(2n) + f(4n) + f(8n) \\
f(16n+4) &= -f(2n) + f(4n) + f(8n+4) \\
f(16n+8) &= -f(2n) + f(4n+3) + f(8n+4) \\
f(16n+12) &= -f(2n) - 2f(4n+2) + 3f(4n+3) + f(8n+4)
\end{aligned}$$

for $n \geq 1$, which allows efficient calculation of this quantity.

5 Linearly Recurrent Sequences

Definition 21. *A recurrent infinite word $\mathbf{x} = a_0 a_1 a_2 \cdots$, where each a_i is a letter, is called* linearly recurrent with constant $L > 0$ *if, for every factor u and its two consecutive occurrences beginning at positions i and j in $\mathbf{x}$ with $i < j$, we have $j - i < L|u|$. The word $a_i a_{i+1} \cdots a_{j-1}$ is called a* return word *of u. Thus linear recurrence can be defined from the condition that every return word w of every factor u of $\mathbf{x}$ satisfy $|w| < L|u|$. Let $\mathcal{R}_u$ denote the set of return words of u in $\mathbf{x}$.*

Remark 22. Linear recurrence implies that every length-k factor appears at least once in every factor of length $(L+1)k - 1$.

Lemma 23 (Durand, Host, and Skau [8]). *Let $\mathbf{x}$ be an aperiodic linearly recurrent word with constant L.*

(i) If u is a factor of $\mathbf{x}$ and w its return word, then $|w| > |u|/L$.
(ii) The number of return words of any given factor u of $\mathbf{x}$ is $\#\mathcal{R}_u \leq L(L+1)^2$.

Theorem 24. *The Lyndon complexity of any linearly recurrent sequence is bounded above by a constant.*

Proof. Let $\mathbf{x}$ be a linearly recurrent sequence with constant L. If $\mathbf{x}$ is ultimately periodic, the subword complexity is already bounded above by a constant, so the Lyndon complexity is also bounded. Now assume that $\mathbf{x}$ is aperiodic, and let $n \geq L$. Denote $k = \lfloor (n+1)/(L+1) \rfloor$, so that

$$(L+1)k - 1 \leq n < (L+1)(k+1) - 1. \tag{1}$$

The left-hand side inequality in (1) and Remark 22 together imply that all factors in $\mathbf{x}$ of length k occur in all factors of length n. Therefore if u is the lexicographically smallest factor of length k, then every Lyndon factor of $\mathbf{x}$ of length n must begin with u. Since every suffix of $\mathbf{x}$ that begins with u can be factorized over $\mathcal{R}_u$, we conclude further that every length-n Lyndon factor of $\mathbf{x}$ is a prefix of a word in $\mathcal{R}_u^*$.

The return words of u have length at least k/L by Lemma 23. Furthermore, the right-hand side inequality in (1) gives

$$\frac{n}{k/L} < \frac{(L+1)(k+1)-1}{k/L} < \frac{L(L+1)(k+1)}{k} \leq 2L(L+1).$$

Therefore any Lyndon factor of length n is a prefix of a word in $\mathcal{R}_u^{2L(L+1)}$. Since $\#\mathcal{R}_u \leq L(L+1)^2$ by Lemma 23, we conclude that

$$\rho_{\mathbf{x}}^L(n) \leq \max\bigl\{\rho_{\mathbf{x}}^L(1), \rho_{\mathbf{x}}^L(2), \ldots, \rho_{\mathbf{x}}^L(L-1), (L(L+1)^2)^{2L(L+1)}\bigr\},$$

so that the Lyndon complexity of $\mathbf{x}$ is bounded.

Definition 25. *Let $h\colon \mathcal{A}^* \to \mathcal{A}^*$ be a primitive morphism, and let $\tau\colon \mathcal{A} \to \mathcal{B}$ be a letter-to-letter morphism. If h is prolongable, so that the limit $h^\omega(a) := \lim_{n\to\infty} h^n(a)$ exists for some letter $a \in \mathcal{A}$, then the sequence $\tau\bigl(h^\omega(a)\bigr)$ is called* primitive morphic.

Lemma 26 (Durand [7,8]). *Primitive morphic sequences are linearly recurrent.*

Corollary 27. *The Lyndon complexity of any primitive morphic sequence is bounded.*

Proof. Follows from Lemma 26 and Theorem 24.

Corollary 28. *If $\mathbf{x}$ is k-automatic and primitive morphic, then its Lyndon complexity is k-automatic.*

Proof. Follows from Corollary 27 and Theorem 13, because a k-regular sequence over a finite alphabet is k-automatic [3].

Acknowledgments. We thank Luke Schaeffer for suggesting the argument in the proof of Theorem 2. We thank the referees for a careful reading of this paper.

References

1. Allouche, J.-P., Rampersad, N., Shallit, J.: Periodicity, repetitions, and orbits of an automatic sequence. Theoret. Comput. Sci. 410, 2795–2803 (2009)
2. Allouche, J.-P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press (2003)
3. Allouche, J.-P., Shallit, J.O.: The ring of k-regular sequences. Theoret. Comput. Sci. 98, 163–197 (1992)
4. Černý, A.: Lyndon factorization of generalized words of Thue. Discrete Math. & Theoret. Comput. Sci. 5, 17–46 (2002)
5. Charlier, É., Rampersad, N., Shallit, J.: Enumeration and Decidable Properties of Automatic Sequences. In: Mauri, G., Leporati, A. (eds.) DLT 2011. LNCS, vol. 6795, pp. 165–179. Springer, Heidelberg (2011)
6. Cobham, A.: Uniform tag sequences. Math. Systems Theory 6, 164–192 (1972)
7. Durand, F.: A characterization of substitutive sequences using return words. Discrete Math. 179, 89–101 (1998)
8. Durand, F., Host, B., Skau, C.: Substitution dynamical systems, bratteli diagrams, and dimension groups. Ergod. Theory & Dynam. Sys. 19, 953–993 (1999)
9. Duval, J.P.: Factorizing words over an ordered alphabet. J. Algorithms 4, 363–381 (1983)
10. Goč, D., Henshall, D., Shallit, J.: Automatic Theorem-Proving in Combinatorics on Words. In: Moreira, N., Reis, R. (eds.) CIAA 2012. LNCS, vol. 7381, pp. 180–191. Springer, Heidelberg (2012)
11. Goč, D., Schaeffer, L., Shallit, J.: The subword complexity of k-automatic sequences is k-synchronized (June 23, 2012) (preprint), `http://arxiv.org/abs/1206.5352`
12. Ido, A., Melançon, G.: Lyndon factorization of the Thue-Morse word and its relatives. Discrete Math. & Theoret. Comput. Sci. 1, 43–52 (1997)
13. Melançon, G.: Lyndon Factorization of Infinite Words. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 147–154. Springer, Heidelberg (1996)
14. Schaeffer, L., Shallit, J.: The critical exponent is computable for automatic sequences. Int. J. Found. Comput. Sci. (2012) (to appear)
15. Séébold, P.: Lyndon factorization of the Prouhet words. Theoret. Comput. Sci. 307, 179–197 (2003)
16. Shallit, J.: The critical exponent is computable for automatic sequences. In: Ambroz, P., Holub, S., Másaková, Z. (eds.) Proceedings 8th International Conference Words 2011. Elect. Proc. Theor. Comput. Sci., vol. 63, pp. 231–239 (2011)
17. Siromoney, R., Mathew, L., Dare, V., Subramanian, K.: Infinite Lyndon words. Inform. Process. Lett. 50, 101–104 (1994)

Efficient Submatch Extraction for Practical Regular Expressions

Stuart Haber[1], William Horne[1,⋆], Pratyusa Manadhata[1], Miranda Mowbray[2], and Prasad Rao[1]

[1] HP Labs Princeton, 5 Vaughn Drive, Suite 301, Princeton, NJ 08540, USA
[2] HP Labs Bristol, Long Down Ave, Stoke Gifford, Bristol BS34 8QT, UK

Abstract. A *capturing group* is a syntax used in modern regular expression implementations to specify a subexpression of a regular expression. Given a string that matches the regular expression, *submatch extraction* is the process of extracting the substrings corresponding to those subexpressions. *Greedy* and *reluctant* closures are variants on the standard closure operator that impact how submatches are extracted. The state of the art and practice in submatch extraction are automata based approaches and backtracking algorithms. In theory, the number of states in an automata-based approach can be exponential in n, the size of the regular expression, and the running time of backtracking algorithms can be exponential in ℓ, the length of the string. In this paper, we present an $O(\ell c)$ runtime automata based algorithm for extracting submatches from a string that matches a regular expression, where $c > 0$ is the number of capturing groups. The previous fastest automata based algorithm was $O(n\ell c)$. Both our approach and the previous fastest one require worst-case exponential compile time. But in practice, the worst case behavior rarely occurs, so achieving a practical speed-up against state-of-the-art methods is of significant interest. Our experimental results show that, for a large set of regular expressions used in practice, our algorithm is approximately twice as fast as Java's backtracking based regular expression library and approximately twenty times faster than the RE2 regular expression engine.

1 Introduction

Regular expressions (REs) are a succinct method to formally represent sets of strings over an alphabet. Given an RE and a string, the RE *matches* the string if the string belongs to the set described by the RE. Many RE implementations also support *search*, i.e. finding the first substring of an input string that matches the RE. In this paper we only address matching, which has practical applications in network security, bioinformatics, and other areas.

Most textbooks on compiler design and related topics (e.g. [4]) describe REs from a theoretical perspective, but omit additional features including *capturing*

⋆ Corresponding author.

groups and *reluctant closure*, that are supported in practical implementations of REs, such as PCRE [3], Java [7], and RE2 [2].

A *capturing group* is a syntax used to specify a subexpression. Given a string that matches the regular expression, *submatch extraction* is the process of extracting the substrings corresponding to those subexpressions. This feature enables regular expressions to be used as parsers. Parentheses are commonly used to indicate the beginning and end of a capturing group. For example, the RE $(.*) = (.*)$ could be used to parse key-value pairs. (Here, the meta-character '.' matches any character in the alphabet, so that $.*$ matches any string.)

The reluctant closure operator, denoted $*?$, appears in both Java and PCRE, and is widely used in practice. This operator is a variant of the standard *greedy closure* operator for REs, denoted $*$, with different submatching behavior: where other rules do not apply, shorter submatches to a subexpression $E*?$ take priority over longer ones, whereas for $E*$ the reverse is true. For example, consider matching the string $a = b = c$ first against $(.*?) = (.*)$, and then against $(.*) = (.*?)$. In the first case the capturing groups match a and $b = c$, respectively, while in the second case the submatches are $a = b$ and c.

If the two closure operators in this example are both greedy or both reluctant, then it is ambiguous which of these two assignments of submatches should be reported by a matching algorithm. There are no formal standards that specify precedence rules for such cases. We aimed for consistency with Java's implementation, which we verified with extensive testing.

Though REs are widely studied, the problem of efficiently implementing submatch extraction has not received much attention. The state of the art includes backtracking and automata based approaches. Java, PCRE, Perl, Python, Ruby, and many other tools implement submatch extraction using recursive backtracking, where an input string may be scanned multiple times before a match is found. Pike implemented the first automata based submatch extraction algorithm in the `sam` text editor [8] based on Thompson's algorithm [10] for RE matching, which converts the RE to a nondeterministic finite automaton (NFA). RE2 uses a combination of deterministic finite automata (DFAs) and NFAs to improve the time efficiency of submatch extraction [2]. RE2 uses DFAs to locate a RE's overall match location in an input string and then uses NFA-based matching on the overall match to extract submatches. Laurikari [5,6] studied ways to implement submatch extraction using a DFA. Both Pike's and Laurikari's implementations require worst-case exponential time to construct the DFA. Once the DFA has been constructed these implementations run in $O(n\ell c)$ time, where n is the number of states in the NFA corresponding to a RE, ℓ is the length of the string being matched, and $c > 0$ is the number of capturing groups.

Our algorithm is suitable for settings where the automaton is compiled once and matched many times against different input strings. This scenario is common, for example, in security applications such as intrusion detection systems and event processing systems which rely heavily on REs and require high-speed matching of input strings.

In this paper, we present an $O(\ell c)$ runtime automata based algorithm for RE matching and submatch extraction. The time complexity of the runtime operation for our algorithm does not depend on n, but our algorithm may require $O(2^n)$ compile time and storage space in the worst case.

However, the asymptotic analysis of these algorithms is deceiving. Backtracking and automata-based approaches almost never have the worst-case behavior on REs that are used in practice. Thus, achieving a practical speed-up against state-of-the-art methods is of significant interest. Our experimental results show that, for a large set of regular expressions used in practice, our algorithm is approximately twice as fast as Java's backtracking based regular expression library and approximately twenty times faster than RE2.

2 Valid Submatch

For the purposes of this paper, the syntax of REs with capturing groups and reluctant closures on an alphabet Σ is

$$E ::= \epsilon \mid a \mid EE \mid E\texttt{|}E \mid E\texttt{*} \mid E\texttt{*?} \mid \texttt{(}E\texttt{)}$$

where a stands for an element of Σ, and ϵ is the empty string. The notation $\texttt{(}E\texttt{)}$ indicates a capturing group. If X, Y are sets of strings we use XY to denote concatenation, i.e. $XY = \{xy : x \in X, y \in Y\}$, and $X|Y$ to denote the union of sets X and Y. If β is a string and B a set of symbols we use $\beta|_B$ to denote the string in B^* obtained by deleting from β all elements that are not in B.

Grouping terms is optional when the order of operations is clear. Specifically, capturing groups have a higher priority than greedy and reluctant closures, which have a higher priority than concatenation, which has a higher priority than union.

We use indices to identify capturing groups within a RE. Given a RE E containing c capturing groups, we assign indices $1, 2, \ldots c$ to each capturing group in the order of their left parentheses as E is read from left to right. We use the notation $idx(E)$ to refer to the resulting *indexed RE*. For example, if $E = ((a){*}|b)(ab|b)$ then $idx(E) = ((a)_2{*}|b)_1(ab|b)_3$.

We introduce the set of symbols $T = \{S_t, E_t : 1 \leq t \leq c\}$, referred to as *tags*, which will be used to encode the start and end of capturing groups.

The language $L(F)$ for an indexed RE $F = idx(E)$ is a subset of $(\Sigma \cup T)^*$, defined by $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$, $L(F_1F_2) = L(F_1) \cdot L(F_2)$, $L(F_1|F_2) = L(F_1) \cup L(F_2)$, $L(F{*}) = L(F{*}?) = L(F){*}$, and $L((F)_t) = \{S_t \alpha E_t : \alpha \in L(F)\}$, where $()_t$ denotes a capturing group with index t.

Definition 1. *A* valid assignment of submatches *for RE E and input string α is a map* SUB : $\{1, \ldots, c\} \to \Sigma^* \cup \{\text{NULL}\}$ *such that there exists $\beta \in L(idx(E))$ satisfying: (i) $\beta|_\Sigma = \alpha$; (ii) if S_t occurs in β then* $\text{SUB}(t) = \beta_t|_\Sigma$*, where β_t is the substring of β between the last occurrence of S_t and the last occurrence of E_t; (iii) if S_t does not occur in β then* $\text{SUB}(t) = \text{NULL}$. □

For example, given the RE $E = ((a){*}|b)(ab|b)$ and an input string $aaab$, the assignment *sub* with $\text{SUB}(1) = aaa$, $\text{SUB}(2) = a$, and $\text{SUB}(3) = b$ is valid for $\beta = S_1S_2aE_2S_2aE_2S_2aE_2E_1S_3bE_3$.

If $\alpha \in \Sigma^*$ we say that α *matches* E if and only if $\alpha = \beta|_\Sigma$ for some $\beta \in L(idx(E))$. Note that for a RE without capturing groups, this coincides with the standard definition of the set of strings matching the expression.

Given a RE containing capturing groups and an input string, the task of a submatch extraction algorithm is to report a valid assignment of submatches if there is one, and to report that the string does not match if there is not.

3 Description of the Algorithm

For this entire section we fix a RE E, and show how to compile E into two deterministic automata, denoted M_3 and M_4, that will be used to match a string. This is done in the preprocessing stage. For the matching and extraction operations, we use M_3 to determine whether the input string matches E, and if it does, we use M_4 to determine what submatches to report.

To understand the need for two automata, consider the RE $(.*)a|(.*)b$. If a procedure is to match a given string against this expression and at the same time to decide whether to match the string to the first or the second capturing group, then clearly the procedure must look ahead to the end of the string. A finite automaton, whose only operating memory is carried by the state it is in, requires other means in order to perform this "look ahead". We achieve this by converting our RE into the DFA M_3 that we run backwards on the input string, accepting or rejecting it as a match; while doing so, we journal the states this DFA goes through. This journaled sequence of states is used as the input to the second automaton, M_4, which has been constructed using tagging information from our RE (encoded by the symbols $S_i, E_i, +, -$). This automaton outputs the appropriate tagging information, which in turn is used to report submatches.

In the preprocessing stage we first construct two other automata, M_1 and M_2, and then use these to derive M_3 and M_4, as described below.

3.1 The Automaton M_1

The automaton M_1 encodes $idx(E)$ as a automaton. We use separate transitions with labels S_t and E_t to indicate the start and end of a capturing group with index t, in addition to transitions labeled with alphabet characters to consume an input character, and transitions labeled with $+$ and $-$ to indicate submatching priorities.

The automaton M_1 is described by the tuple $(Q_1, \Sigma_1, \Delta_1,\ s_1, f_1)$, where Q_1 is a set of states identified by the integers in the set $\{1, 2 \ldots f\}$, Σ_1 is the alphabet $\Sigma \cup \{+, -\} \cup T$, where $+$ and $-$ are two special alphabet characters that will be described below, Δ_1 is a transition function, $s_1 = 1$ is the start state and $f_1 = f$ is the unique final state. Δ_1 is built using structural induction on $idx(E)$ following the rules illustrated by the diagrams in Fig. 1. The initial state is marked with $>$ and the final state with a double circle. A dashed arrow with label F or G is used as shorthand for the diagram corresponding to the indexed

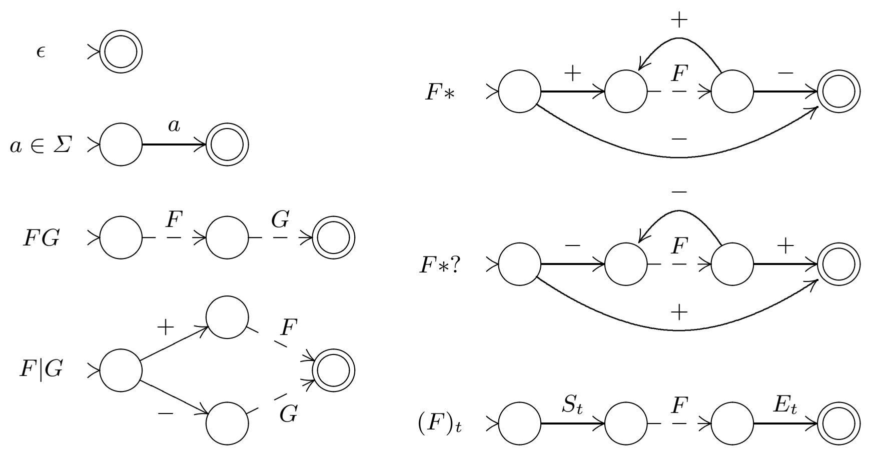


Fig. 1. Rules for the construction of M_1

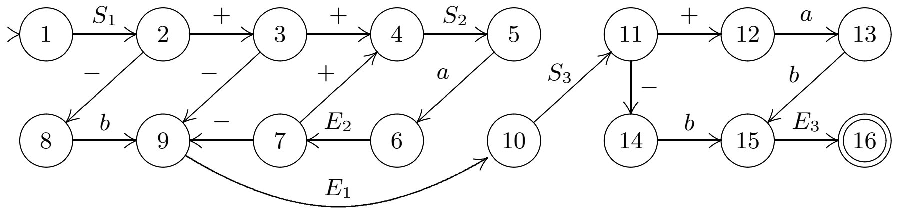


Fig. 2. The DFA M_1 for $((a)*|b)(ab|b)$

expression F or G. For example, the automaton M_1 for $((a)*|b)(ab|b)$ is shown in Fig. 2.

If x is any directed path in M_1, when it is considered as a directed graph, we write $ls(x)$ for its label sequence. For example in Fig. 2 $ls(8 \to 9 \to 10 \to 11 \to 14) = bE_1S_3-$.

Let $\pi : Q_1 \times Q_1 \to T^*$ be a mapping from a pair of states to a sequence of tags, to be used in the constructions below, defined as follows. For any two states $p, q \in Q_1$, consider a depth-first search of the graph of M_1, beginning at p and searching for q, using only transitions with labels from $T \cup \{+, -\}$. The construction rules for M_1 ensure that if there is any state with two different outgoing transitions, one will be labeled '$+$' and the other '$-$'. The search explores all states reachable via the transition labeled '$+$' before following the one labeled '$-$'. If this search succeeds, finding successful search path $\lambda(p, q)$, then $\pi(p, q) = ls(\lambda(p, q))|_T$ is the sequence of tags along this path. If it fails, then $\pi(p, q)$ is undefined. Note

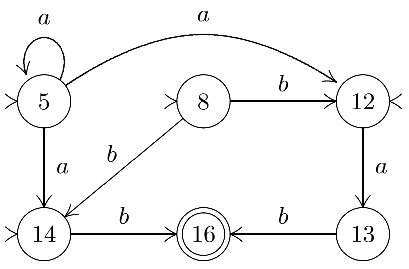


Fig. 3. The automaton M_2 for the RE $((a)*|b)(ab|b)$

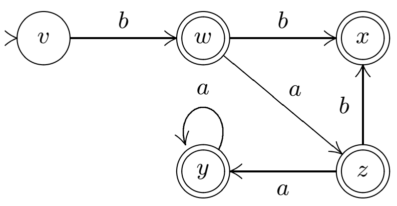


Fig. 4. The automaton M_3 for for the RE $((a)*|b)(ab|b)$, where $v = \{16\}$, $w = \{13, 14\}$, $x = \{8\}$, $y = \{5\}$, and $z = \{5, 12\}$

that $\pi(p, p)$ is defined to be the empty string, and that this description of the search uniquely specifies $\lambda(p, q)$, if it exists.

3.2 The Automaton M_2

Next, we convert M_1 into another automaton, the NFA M_2, described by the tuple $(Q_2, \Sigma, \Delta_2, S_2, f)$. The set Q_2 consists of the final state of M_1 together with any state in M_1 that has an outgoing transition labeled with a symbol in Σ, i.e.

$$Q_2 = \{f\} \cup \{q : \exists a \in \Sigma, p \in Q_1, (q, a, p) \in \Delta_1\}$$

If $p, q \in Q_2$ and $a \in \Sigma$, there is a transition $(p, a, q) \in \Delta_2$ if and only if there exists a state $r \in Q_1$ such that $(p, a, r) \in \Delta_1$ and $\pi(r, q)$ is defined. S_2 is a set of initial states, corresponding to those states, $p \in Q_2$, for which $\pi(1, p)$ is defined.

For example, the automaton M_2 for $((a)*|b)(ab|b)$ is shown in Fig. 3.

3.3 The Automaton M_3

Next, we convert M_2 into the DFA M_3, specified by the tuple $(Q_3, \Sigma, \Delta_3, \; s_3, F_3)$. The construction of M_3 from M_2 is a standard powerset construction of a DFA from a reversed NFA [9], modified in order to process the input string backwards. Specifically, each state in Q_3 corresponds to a subset of states in the powerset of Q_2. The initial state s_3 is $\{f\}$. We initialize Q_3 to $\{\{f\}\}$, and recursively add states r to Q_3 by constructing for each $a \in \Sigma$ the set

$$P(r, a) = \{p \in Q_2 : (p, a, q) \in \Delta_2 \text{ for some } q \in r\},$$

i.e. the set of states from which there is a transition labeled a to an element of r. If this set is not empty, it is added to Q_3 and the transition $(r, a, P(r, a))$ is added to Δ_3. We explore each previously unexplored state in Q_3 until there are no states in Q_3 left to explore. The set of final states in M_3, F_3, consists of any state q in Q_3 such that $q \cap S_2$ is not empty. The DFA M_3 for $((a)*|b)|(ab|b)$ is shown in Fig. 4.

3.4 The Automaton M_4

Next, we use M_1, M_2 and M_3 to construct another automaton, M_4, described by the tuple $(Q_4, \Sigma_4, \Delta_4, s_4)$. Q_4 is essentially M_2 with one extra state, where the input alphabet is $\Sigma_4 = Q_3$ instead of Σ, and some edges are deleted. Specifically, we introduce a new state labeled '0', which will be the start state of M_4, so that $Q_4 = Q_2 \cup \{0\}$. This is a DFA except that the transition function is a four-tuple, i.e. $\Delta_4 \subseteq Q_2 \times Q_3 \times Q_2 \times T^*$.

The definition of M_4 uses a partial ordering on label sequences of paths in M_1 that corresponds to the priorities for submatches. The intuition for M_4 is that a transition (p, Q, q, τ) of M_4 exists if, among all the paths in M_1 that have start state p, end state in Q, first label in Σ and no other labels in Σ, the path with the highest-priority label sequence ends at state q and has label sequence $a\tau$ for some alphabet symbol $a \in \Sigma$. The output τ encodes the capturing groups that are entered and left as this path is followed; during the runtime operation, this information will be used to determine the submatch that should be reported for each capturing group.

Let $\prec$ be the lexicographic partial ordering on Σ_1^* that is induced by the relation $\{(a, a) : a \in \Sigma_1\} \cup \{(-, +)\}$ on Σ_1. For example, if a, b, c are different elements of Σ, then $a \prec a{-}{+}b \prec a{+}c$, but $ab \not\prec ac$ and $ac \not\prec ab$. Finally, we define Δ_4, the transition function for M_4, as follows. Let (p, Q, q, τ) be in Δ_4 iff there exist $p, r \in Q_2$, $Q \in Q_3$, $q \in Q$, $a \in \Sigma$, such that $(p, a, r) \in \Delta_1$, $\pi(r, q)$ is defined, and

$$\tau = \pi(r, q) = (\max_{\prec}\{ls(\lambda(r, q')) \ : \ q' \in Q\}) \mid_T .$$

Similarly, let $(0, Q, q, \tau)$ be in Δ_4 iff there exist $Q \in Q_3$, $q \in Q$ such that $\pi(1, q)$ is defined, and

$$\tau = \pi(1, q) = (\max_{\prec}\{ls(\lambda(1, q')) \ : \ q' \in P\}) \mid_T .$$

For space considerations, we omit from this paper the proof that the maximal elements used in these definitions exist, and are unique. Continuing with our running example, the automaton M_4 for $((a)*|b)(ab|b)$ is shown in Fig. 5.

3.5 Runtime Operation

We extract submatches for a string $a_1 \ldots a_\ell \in \Sigma^*$ in runtime in three steps:

1. We process the string $a_\ell a_{\ell-1} \ldots a_1$ using M_3. As it is processed, we journal the states $q_\ell, q_{\ell-1}, \ldots$ visited during the processing, where q_ℓ is $\{f\}$, the initial state of M_3. If the processing terminates before the whole input string has been processed (i.e. because we hit a "dead state" of M_3), or terminates with $q_0 \notin F_3$, we report that the string does not match and stop. This step runs in $O(\ell)$ time.
2. If we did not stop in the previous step, we run M_4 on input $q_0, q_1, \ldots q_\ell$, using an additional data structure along the way in order to discover the

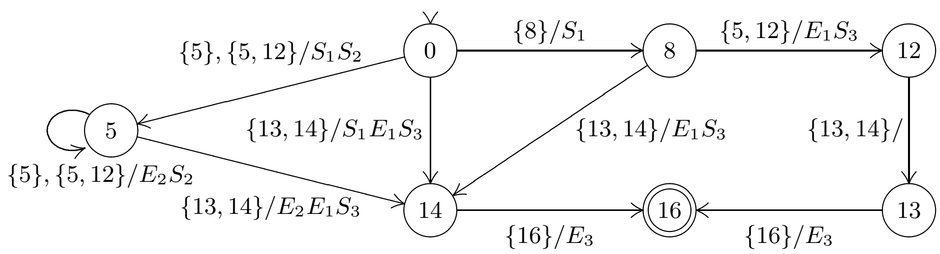


Fig. 5. The automaton M_4 for the RE $((a)*|b)(ab|b)$. Transitions are labeled with a slash separating inputs from outputs.

submatch values for each capturing group. The data structure consists of an array of length $2c$, indexed by elements of T, all initialized to NULL. While processing the ith transition, namely $(q_i, P, q_{i+1}, \tau) \in \Delta_4$, for each tag in $\tau \in T^*$ we write i in the array entry corresponding to the tag, overwriting the current entry. This step runs in $O(\ell c)$ time.

3. We use the resulting array to read off the submatches from the input string, as follows. If the array entries for the tags S_j and E_j are s_j and e_j, respectively, then the submatch for capturing group j is $a_{s_j+1} \dots a_{e_j}$. If the array entries S_j and E_j are NULL, then there is no submatch for the jth capturing group. This step runs in $O(\ell c)$ time.

The first two steps together are called *matching*; the third step is called *extraction*. For example, consider processing the input string $aaab$ for the RE $((a)*|b)|(ab|b)$. In step 1, we process the string $baaa$ with M_3. The states visited are $\{16\}$, $\{13, 14\}$, $\{5, 12\}$, $\{5\}$, $\{5\}$ (see Fig. 4). In step 2, we run automaton M_4 with input $\{5\}, \{5\}, \{5, 12\}, \{13, 14\}, \{16\}$, writing entries in the array with each transition (see Fig. 5). The resulting array reads

$$[S_1, E_1, \quad S_2, E_2, \quad S_3, E_3] = [0, 3, \quad 2, 3, \quad 3, 4].$$

In step 3, we read from the array that the three capturing groups have respective submatches aaa, a, and b.

To see that the $O(\ell c)$ complexity bound in step 2 gives the worst-case runtime for our algorithm, suppose that $E = [a(F_1)(F_2)...(F_c)]^*$ (using square brackets to denote a non-capturing group) with $a, b \in \Sigma$, $F_1 = b|\epsilon$, $F_2 = bb|\epsilon$, $F_3 = bbb|\epsilon \dots$, and $a_i = a$ for all $1 \leq i \leq \ell$. Then for $1 \leq i \leq \ell$, the string output by M_4 when processing q_i in the second step of the operation is $S_1E_1S_2E_2 \dots S_cE_c$, and so in this case the operation updates $2\ell c$ array elements.

Note that in our analysis, we assume that we can read or write the index of one of the states of our automata in constant time and space. In practice, we never run our algorithm for a RE whose automaton is exponentially large.

3.6 Correctness

Here we prove the theorem, which shows the correctness of our algorithm.

Theorem 2. *Suppose that the input string $\alpha = a_1 \dots a_n$ matches the RE E. Then our algorithm reports a valid assignment of submatches for E and α.*

Proof. We will prove the theorem by constructing a string $\gamma \in L(E)$, and then showing that the assignment of submatches for E and α satisfies the three properties in Definition 1 for a valid assignment, with β equal to γ.

The first step is to show that the operation of our algorithm does not terminate before all of $q_0, q_1, \dots q_\ell$ have been processed by M_4. Since α matches E, it is accepted by M_2, and by Rabin and Scott's result relating languages of automata [9] this implies that the reverse of α is accepted by M_3; thus, the first step of the runtime operation does not terminate early, and ends at state q_0 in F_3. By definition of F_3, there is some $j \in q_0$ such that $\pi(1, j)$ is defined. Therefore $(0, q_0, j_0, \tau_0) \in \Delta_4$ for some $j_0 \in q_0$ and $\tau_0 = \pi(1, j_0) \in T^*$. So the processing of $q_0, q_1 \dots q_\ell$ by M_4 does not terminate before q_0 has been processed.

Suppose inductively that $1 \leq i \leq \ell$, the processing of $q_0 \dots q_\ell$ by M_4 does not terminate before q_{i-1} has been processed, and that $j_{i-1} \in q_{i-1}$, where j_{i-1} is the state of M_4 reached just after q_{i-1} has been processed. Now, q_i is the i^{th} state of Q_3 visited when $a_\ell a_{\ell-1} \dots a_1$ is processed by M_3, and so it is the set of elements of Q_2 from which there is a path in M_2 from q to f with label sequence $a_{i+1} \dots a_\ell$. Therefore, $(j_{i-1}, a_i, j) \in \Delta_2$ for some $j \in q_i$. By the definition of M_2, there is some $k_i \in Q_1$ such that $e_i = (j_{i-1}, a_i, k_i) \in \Delta_1$ and $\pi(k_i, j)$ is defined. By the construction of M_4, it follows that there is some $j_i \in q_i$ and $\tau_i = \pi(k_i, j_i) \in T^*$ such that $(j_{i-1}, q_i, j_i, \tau_i) \in \Delta_4$. This shows that for $1 \leq i \leq \ell$, q_i is processed in step 2 of the operation, as required.

Note that $j_\ell \in q_\ell = \{f\}$, so $j_\ell = f$. Let y be the path in M_1 from 1 to f obtained by concatenating $\lambda(1, j_0)$, e_1, $\lambda(k_1, j_1)$, $\dots$ e_ℓ, $\lambda(k_\ell, j_\ell)$. Now we can define γ: it is $ls(y)|_{\Sigma \cup T}$. Note that it is equal to the concatenation of τ_0, a_1, τ_1, $\dots$ a_ℓ, τ_ℓ.

It is straightforward to prove by induction on the size of E that $L(E) = \{\beta|_{\Sigma \cup T} : M_1 \text{ accepts } \beta\}$. The automaton M_1 accepts $ls(y)$, so $\gamma \in L(E)$.

We will now show that the assignment of submatches reported by the operation satisfies the three criteria for a valid assignment, with β equal to the string γ. Property (i) holds because $\gamma|_\Sigma = a_1 \dots a_\ell = \alpha$.

For property (ii), observe that for $0 \leq i \leq \ell$, when q_i is processed in step 2 of the operation, i is written in the array entry for each tag in τ_i. Thus if the array entries for S_j and E_j at the end of step 2 are s_j and e_j respectively, then the last occurrence of S_j in γ lies before a_1 if $s_j = 0$, between a_{s_j} and a_{s_j+1} if $0 < s_j < \ell$ or after a_ℓ if $s_j = \ell$, and similarly for e_j and the last occurrence of E_j in γ. Property (ii) follows.

For property (iii), observe that it follows from the definition of $L(E)$ that if S_t occurs in a string in $L(E)$, then E_t must also occur in the string. Suppose that S_t does not occur in γ. Then neither does E_t, and so S_t, E_t do not occur in any of $\tau_0, \dots \tau_\ell$. So at the end of step 2 the array entries for S_t, E_t are NULL, and step 3 reports that there is no match to capturing group t, as required.

Table 1. Results for DHCP and Snort log experiments (times in microseconds)

RE	Our Algorithm			Java			RE2	
	compile	match	extract	compile	match	extract	compile	match+extract
DHCP	16,993	47,685	19,809	44	119,852	21,423	32	1,153,118
Snort	24,761	64,484	804	14	138,138	1,666	22	1,414,001

4 Evaluation

Our first set of experiments deals with parsing logs for Microsoft DHCP logs and for Snort, which is an open source intrusion detection system. DHCP logs are a simple comma-separated format with exactly eight fields. To parse such a log record, the following RE is used,

```
([^,]*),([^,]*),([^,]*),([^,]*),([^,]*),([^,]*),([^,]*),([^,]*)
```

where the syntax `[^,]` means any character except a comma.

For Snort logs, we must extract the source and destination IP addresses and the source and destination ports (if they exist). The RE we used is

```
.*? (\d+\.\d+\.\d+\.\d+)(:\d+)? -> (\d+\.\d+\.\d+\.\d+)(:\d+)? .*
```

where the metacharacter `\d` represents any numeral, the operator `+` is a variation on closure that requires at least one instance of its operand, and the operator `?` means exactly zero or one instances of its operand.

All experiments were performed on a workstation with twelve 2.67GHz cores and 6GB of RAM. Our algorithm was implemented in Java, whereas RE2 is implemented in C++.

We ran an experiment where we matched 100,000 lines of DHCP log files and 25,741 lines of Snort log files against the regular expressions and experimentally evaluated it in comparison to Java and RE2, as shown in Table 1. Since RE2 matches and extracts in a single operation, these are grouped in the table. Clearly, our algorithm is much slower in the compilation phase than either Java or RE2. But as we have discussed in the introduction, we are willing to incur a significant penalty in the compilation phase, since for the problems we are interested in, we perform compilation once, but matching and extraction many times for each RE. When we amortize the compilation time over the matching and extraction time with multiple strings, our algorithm can actually outperform Java and RE2. This result is surprising given that RE2 has previously been reported to be faster than other `C/C++` based RE engines [1].

Next, we ran several experiments evaluating our algorithm for performance, storage usage, and correctness on a set of REs that are included as part of a commercial Security Information and Event Management (SIEM) system that uses 16,805 unique REs. Of these REs, 7732 (46.0%) have no capturing groups, and thus can be matched with an ordinary DFA; 7596 (45.2%) can be implemented with our two pass algorithm; 1396 of these REs (8.3%) can be implemented more

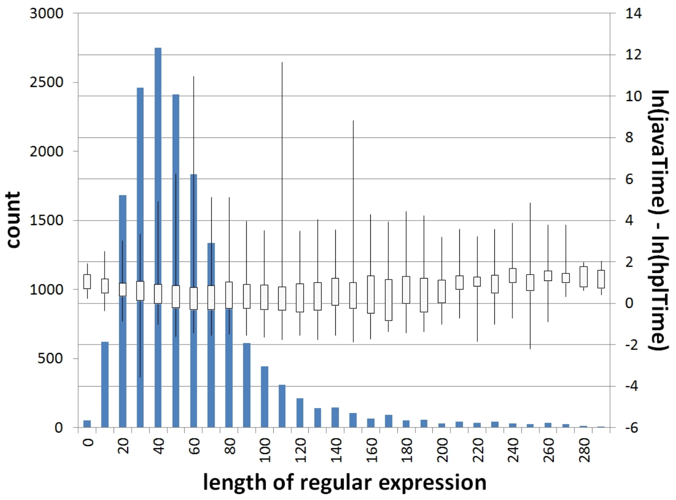


Fig. 6. Performance as a function of RE length

efficiently using a variation on our algorithm that only requires one pass (the details of that algorithm are beyond the scope of this paper); 51 (0.3%) caused the size of M_3 in our algorithm to grow beyond 4096 states, at which point we declared failure; 20 (0.1%) have syntactic features which we have not yet implemented in our algorithm, but which we believe will not impact performance.

For the first four categories of REs, we synthetically generated 1,000 matching strings for each of the REs. We then measured the time to match those strings using both our algorithm and Java. The results (broken down by RE length) are shown in Fig. 6. The blue bars are a histogram of the number of REs that have a length in each bin range. The count of the number of REs is shown on the left-hand y-axis. We then measure the performance as the log of the time taken by Java minus the log of the time taken by our algorithm in order to show speedups in both directions symmetrically. We then plot those results as a box plot showing the first, second, third and fourth quartiles for each bin. The min and max performance are the end of the line segments, while the range between the second and third quartile is shown in the box. The range of performance values are shown on the right-hand y-axis. As can be seen in the figure, our algorithm is faster than Java most of the time, regardless of the length of the RE, often significantly faster. On average, we are 2.3 times faster than Java.

This was a computationally intensive test which took over 8 hours on our workstation at 580,694 matches per second: we performed 1,000 tests on 1,000 strings for 16,724 REs. We omitted the performance comparisons against RE2 because RE2 would simply take too long.

Theoretically, the number of states in a DFA built using a powerset construction could be exponential in the size of the RE. However, as described above, less than 0.3% of the REs exhibited such behavior. In fact, for 99% of the DFAs built with a powerset construction, the ratio of the number of states to the length of the RE string was less than 5.25. For approximately 58% of the REs, the DFA actually had fewer states than the length of the RE string. The average RE needed 28 kBs for the transition tables and other associated data structures.

We were able to measure the memory usage of our own algorithm, but accurately measuring the data structures associated with regular expressions is infeasible in third party software. Regardless, backtracking algorithms generally just need enough space to store the regular expression itself, which is essentially negligible.

Although we have proved that our algorithm is guaranteed to generate a valid assignment of submatches, we are particularly interested in showing that our algorithm generates the same submatches as Java since there may be multiple valid assignments of submatches for a given RE. We synthetically generated 15 matching and non-matching strings for each RE in the first four categories. The submatches extracted by our approach and Java were identical.

5 Summary

In this paper, we introduced a new algorithm for converting REs to automata that handles submatch extraction and reluctant closures. Our experimental results show that our algorithm is approximately twice as fast as Java's backtracking based regular expression library and approximately twenty times faster than RE2 on real-world REs used for several problems involving processing of security event logs, including a comprehensive test of the algorithm against a database of 16,724 REs used by a commercial SIEM system.

References

1. Benchmark of Regex Libraries (July 2010), `http://lh3lh3.users.sourceforge.net/reb.shtml`
2. RE2 (January 2012), `http://code.google.com/p/re2/`
3. PCRE (2011), `http://www.pcre.org/`
4. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley (2003)
5. Laurikari, V.: NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In: Proc. of the 7th Int. Symp. on String Processing and Information Retrieval, pp. 181–187 (2000)
6. Laurikari, V.: Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology (2001)
7. Nourie, D., McCloskey, M.: Regular Expressions and the Java Programming Language (2010), `http://java.sun.com/developer/technicalArticles/releases/1.4regex`
8. Pike, R.: The Text Editor sam. Softw. Pract. Exper. 17, 813–845 (1987)
9. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Research and Development 3(2) (April 1959), doi:10.1147/rd.32.0114
10. Thompson, K.: Programming techniques: Regular expression search algorithm. Comm. ACM 11, 419–422 (1968)

Determinacy and Subsumption for Single-Valued Bottom-Up Tree Transducers

Kenji Hashimoto[1], Ryuta Sawada[2], Yasunori Ishihara[2], Hiroyuki Seki[1], and Toru Fujiwara[2]

[1] Nara Institute of Science and Technology, Japan
{k-hasimt,seki}@is.naist.jp
[2] Osaka University, Japan
{ishihara,fujiwara}@ist.osaka-u.ac.jp

Abstract. This paper discusses the decidability of determinacy and subsumption for tree transducers. For two tree transducers T_1 and T_2, T_1 determines T_2 if the output of T_2 is identified by the output of T_1, that is, there is a partial function f such that $[\![T_2]\!] = f \circ [\![T_1]\!]$ where $[\![T_1]\!]$ and $[\![T_2]\!]$ are tree transformation relations induced by T_1 and T_2, respectively. Also, T_1 subsumes T_2 if T_1 determines T_2 and the partial function f such that $[\![T_2]\!] = f \circ [\![T_1]\!]$ can be defined by a transducer in a designated class that T_2 belongs to. In this paper, we show that determinacy is decidable for single-valued linear extended bottom-up tree transducers as the determiner class and single-valued bottom-up tree transducers as the determinee class. We also show that subsumption is decidable for these classes.

1 Introduction

In data transformation, it is desirable that certain information in source data be preserved through transformation. As a formalization for information preservation in data transformation, the notions of *deteminacy* and *subsumption* (or query rewriting) are known [1,6,11]. Let Q be a query to a database and V be a data transformation (or a view definition) of the database. Determinacy of Q by V means that the answer to Q is identified by the answer to V. When information to be preserved is specified by a query Q, determinacy guarantees that for any database instance D, $V(D)$ gives enough information to uniquely determine the specified information $Q(D)$ for D. Subsumption means that the answer to Q can always be computed from the answer to V by some query in a designated class that Q belongs to. Compared with determinacy, subsumption guarantees that the necessary information $Q(D)$ can be extracted from the transformed data $V(D)$ by the same query language expressing Q.

We study the decidability of determinacy and subsumption when both a query and a data transformation are given by tree transducers. Tree transducers are machines that model relations between labeled ordered trees. A tree transducer is said to be *single-valued* if the tree transformation induced by the transducer is a partial function. Since an XML document has a tree structure, tree transducers

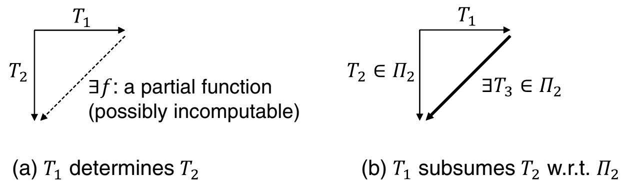

T_1
T_2
$\exists f$: a partial function (possibly incomputable)
(a) T_1 determines T_2
T_1
$T_2 \in \Pi_2$
$\exists T_3 \in \Pi_2$
(b) T_1 subsumes T_2 w.r.t. Π_2

Fig. 1. Determinacy and subsumption

are often used as a model of XML document transformations. Formally, for two single-valued tree transducers T_1 and T_2 in classes Π_1 and Π_2 of transducers, respectively, we say T_1 *determines* T_2 if there is a partial function f such that $[\![T_2]\!] = f \circ [\![T_1]\!]$ (see Fig. 1(a)), where $[\![T_1]\!]$ and $[\![T_2]\!]$ are the tree transformation relations induced by T_1 and T_2, respectively. Π_1 and Π_2 are called the determiner class and the determinee class, respectively. We also say T_1 *subsumes* T_2 with respect to Π_2, if T_1 determines T_2 and the partial function f such that $[\![T_2]\!] = f \circ [\![T_1]\!]$ can be defined by a transducer in the class Π_2 (see Fig. 1(b)). Our goal is to find practical subclasses of tree transducers for which determinacy and subsumption are decidable, and to consider the problem of constructing a tree transducer T_3 in the determinee class such that $[\![T_2]\!] = [\![T_3]\!] \circ [\![T_1]\!]$ if T_1 subsumes T_2.

In this paper, we first show that determinacy is decidable for single-valued linear extended bottom-up tree transducers (sl-xbots) as the determiner class and single-valued bottom-up tree transducers (s-bots) as the determinee class running over a ranked-tree encoding of the given XML document. Transformations induced by transducers in the classes include simple filterings, relabelings, insertions, and deletions of elements. Especially, sl-xbots do not allow duplications of elements. For some other classes, we show that determinacy is undecidable even for homomorphism tree transducers as the determiner class, which is a proper subclass of s-bots and single-valued top-down tree transducers (s-tops). Moreover, determinacy is undecidable for deterministic monadic second-order logic defined tree transducers (dmsotts) [2,5] as the determiner class, which form a class incompatible with s-bots and s-tops but is a proper superclass of sl-xbots. Lastly, we show that subsumption is decidable for sl-xbots as the determiner class and s-bots as the determinee class. The proof gives a construction method of an s-bot T_3 satisfying $[\![T_2]\!] = [\![T_3]\!] \circ [\![T_1]\!]$ if T_1 subsumes T_2.

Due to space limitation, we omit most of the proofs including details of some construction methods. They are presented in the full version of this paper [8].

2 Preliminaries

2.1 Trees and Tree Automata

We denote the set of nonnegative integers by $\mathbb{N}$. Let $[i, j] = \{d \in \mathbb{N} \mid i \leq d \leq j\}$. In particular, we denote $[1, k]$ by $[k]$. A (ranked) alphabet is a finite set Σ of symbols with a mapping rk from Σ to $\mathbb{N}$. We denote the set of k-ary symbols of Σ by $\Sigma^{(k)} = \{\sigma \in \Sigma \mid \mathsf{rk}(\sigma) = k\}$. The set $\mathcal{T}_\Sigma$ of *ranked trees* over an alphabet Σ is the smallest set T such that $\sigma(t_1, \ldots, t_k) \in T$ for every $k \in \mathbb{N}$, $\sigma \in \Sigma^{(k)}$, and $t_1, \ldots, t_k \in T$. If $\sigma \in \Sigma^{(0)}$, we write σ instead of $\sigma()$. The set of *positions* of $t = \sigma(t_1, \ldots, t_k) \in \mathcal{T}_\Sigma$, denoted by $pos(t)$, is defined by $pos(t) = \{\epsilon\} \cup \{ip \mid i \in [k], p \in pos(t_i)\}$ where $\sigma \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$. We write $p \preceq p'$ when p is a prefix of p', that is, p is an ancestor position of p', and $p \prec p'$ when p is a proper prefix of p'. For $p, p' \in pos(t)$, let $\mathrm{nca}(p, p')$ be the nearest common ancestor position of p and p', that is, the longest common prefix of p and p'. For $p \in pos(t)$, $t|_p$ denotes the subtree of t at p, and $t[t']_p$ denotes the tree obtained from t by replacing the subtree at p with t'.

Let $X = \{x_*\} \cup \{x_i \mid i \geq 1\}$ be a set of variables of rank 0, and for every $k \geq 1$, $X_k = \{x_i \mid i \in [k]\}$. For $V \subseteq X$, we often write $\mathcal{T}_\Sigma(V)$ to mean $\mathcal{T}_{\Sigma \cup V}$. A tree $t \in \mathcal{T}_\Sigma(V)$ is *linear* if each variable in V occurs at most once in t. Let $C_\Sigma(V)$ denote the set of linear trees in $T_\Sigma(V)$. Let $\bar{\mathcal{T}}_\Sigma(V)$ (resp. $\bar{\mathcal{C}}_\Sigma(V)$) be the set of trees in $\mathcal{T}_\Sigma(V)$ (resp. $\mathcal{C}_\Sigma(V)$) such that each variable in V occurs at least once. Note that $\bar{\mathcal{T}}_{\Sigma \cup V}(V')$ denotes the set of trees in $\mathcal{T}_\Sigma(V \cup V')$ such that every variable in V' must occur at least once. For $t \in \mathcal{T}_\Sigma(X)$ and $\sigma \in \Sigma \cup X$, let $pos_\sigma(t)$ be the set of the positions of t at which σ occurs, and $pos_Y(t) = \bigcup_{\sigma \in Y} pos_\sigma(t)$ for $Y \subseteq \Sigma \cup X$. Let $\mathrm{var}(t)$ be the set of variables of t, and $\mathrm{yield}_X : \mathcal{T}_\Sigma(X) \to X^*$ be the function such that $\mathrm{yield}_X(x) = x$ for every $x \in X$ and $\mathrm{yield}_X(\sigma(t_1, \ldots, t_k)) = \mathrm{yield}_X(t_1) \cdots \mathrm{yield}_X(t_k)$ for every $\sigma \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in \mathcal{T}_\Sigma(X)$. A tree $t \in \mathcal{T}_\Sigma(X)$ is *normalized* if $\mathrm{yield}_X(t) = x_1 \cdots x_k$ for some $k \in \mathbb{N}$. Every mapping $\theta : V \to \mathcal{T}_\Sigma(X)$ with $V \subseteq X$ is called a substitution, which can be extended to $\theta : \mathcal{T}_\Sigma(V) \to \mathcal{T}_\Sigma(X)$. If $V = X_k$ and $x_i\theta = t_i$ for each $i \in [k]$, we also denote $t\theta$ by $t[t_1, \ldots, t_k]$, and if $V = \{x\}$ and $\theta(x) = t'$, we denote $t\theta$ by $t[x \leftarrow t']$. In particular, if $V = \{x_*\}$ and $\theta(x_*) = t'$, we denote $t\theta$ by $t[t']$ or simply tt'.

A *finite tree automaton* (TA for short) is a 4-tuple $A = (Q, \Sigma, Q_a, \gamma)$, where Q is a finite set of states, Σ is an alphabet, $Q_a \subseteq Q$ is a set of accepting states, and γ is a finite set of transition rules, each of which is of the form $(q, C[q_1, \ldots, q_k])$ where $q, q_1, \ldots, q_k \in Q$ and $C \in \bar{\mathcal{C}}_\Sigma(X_k)$. The move relation $\Rightarrow_A$ of a TA $A = (Q, \Sigma, Q_a, \gamma)$ is defined as follows: if $(q, C[q_1, \ldots, q_k]) \in \gamma$ and $t|_p = C[q_1, \ldots, q_k]$ where $p \in pos(t)$, then $t \Rightarrow_A t[q]_p$. Let $L(A)$ denote $\{t \mid t \Rightarrow_A^* q_a, q_a \in Q_a\}$ where $\Rightarrow_A^*$ is the reflexive transitive closure of $\Rightarrow_A$. For a state q of A, let $A(q)$ be a TA obtained from A by replacing the set Q_a of accepting states with the singleton $\{q\}$. A set L of trees such that $L = L(A)$ for some TA A is called a regular tree language, or we say L is regular.

2.2 Tree Transducers

An *extended bottom-up tree transducer* (xbot) [4] is a 5-tuple $(Q, \Sigma, \Delta, Q_f, \delta)$, where Q is a finite set of states, Σ is an input alphabet, Δ is an output alphabet, $Q_f \subseteq Q$ is a set of final states, and δ is a set of transduction rules of the form $C_l[q_1(x_1), \ldots, q_k(x_k)] \to q(t_r)$ where $k \in \mathbb{N}$, $C_l \in \bar{\mathcal{C}}_\Sigma(X_k)$, $t_r \in \mathcal{T}_\Delta(X_k)$, $q, q_1, \ldots, q_k \in Q$. A rule is normalized if its left-hand side is normalized. Without loss of generality, we can assume that every rule is normalized. A rule $\rho \in \delta$ is an *ϵ-rule* if the left-hand side of ρ is the form $q(x)$ where $q \in Q$ and $x \in X$, and it is *input-consuming* otherwise. Let $T = (Q, \Sigma, \Delta, Q_f, \delta)$ be an xbot. T is a *bottom-up tree transducer* (bot) if the left-hand side of every rule in δ contains exactly one symbol in Σ. Also, we denote by an xbot^{-e} an xbot without ϵ-rules. T is a *linear extended bottom-up tree transducer* (l-xbot) if the tree t_r in the right-hand side of each rule in δ is linear.

The move relation $\Rightarrow_T$ of an xbot $T = (Q, \Sigma, \Delta, Q_f, \delta)$ is defined as follows: $t \Rightarrow_T^{\rho} t'$ for a rule $\rho = (C_l[q_1(x_1), \ldots, q_k(x_k)] \to q(t_r)) \in \delta$ if there exists a position $p \in pos(t)$ such that $t|_p = C_l[q_1(t_1), \ldots, q_k(t_k)]$ where $t_1, \ldots, t_k \in \mathcal{T}_\Delta(X)$ and $t' = t[q(t_r[t_1, \ldots, t_k])]_p$, and $t \Rightarrow_T t'$ if there exists $\rho \in \delta$ such that $t \Rightarrow_T^{\rho} t'$. The transformation *induced* by T, denoted as $[\![T]\!]$, is the relation defined as $\{(t, t') \mid t \Rightarrow_T^* q_f(t'), t \in \mathcal{T}_\Sigma, t' \in \mathcal{T}_\Delta, q_f \in Q_f\}$ where $\Rightarrow_T^*$ is the reflexive transitive closure of $\Rightarrow_T$. The domain of T, denoted by $\mathrm{dom}(T)$, is $\{t \mid (t, t') \in [\![T]\!]\}$, and the range of T, denoted by $\mathrm{rng}(T)$, is $\{t' \mid (t, t') \in [\![T]\!]\}$. For a tree t, $[\![T]\!](t) = \{t' \mid (t, t') \in [\![T]\!]\}$. For a TA A, the image $T(A)$ of $L(A)$ by T is $\{t' \mid (t, t') \in [\![T]\!], t \in L(A)\}$. For a state q of T, let $T(q)$ be an xbot obtained from T by replacing the set Q_f of final states with the singleton $\{q\}$.

The tree transducers T and T' are *equivalent* if $[\![T]\!] = [\![T']\!]$. For tree transducers T_1 and T_2, $[\![T_2]\!] \circ [\![T_1]\!] = \{(t, t') \mid (t, t'') \in [\![T_1]\!], (t'', t') \in [\![T_2]\!]\}$. A transducer T is said to be *single-valued* (or *functional*) if any two pairs of (t, t') and (t, t'') in $[\![T]\!]$ satisfy $t' = t''$. We denote the unique output tree of T on a tree t by $T(t)$. It is known that the single-valuedness of bots is decidable in polynomial time [9]. We use the prefix 's' to represent that a transducer is single-valued, e.g., we write for short an s-xbot to denote a single-valued xbot.

Without loss of generality, we assume that any alphabet contains a special symbol $\perp$, which means "no output" and does not occur in any final output tree. We recall the notion of reducedness [9], which is defined for bots but can be naturally applied to xbots. An xbot $T = (Q, \Sigma, \Delta, Q_f, \delta)$ is called *reduced* if and only if the following two conditions hold:

1. T has no useless states, that is, for every state $q \in Q$, there exists a tree $t = Ct_s \in \mathrm{dom}(T)$ where $C \in \bar{\mathcal{C}}_\Sigma(\{x_*\})$ such that $t \Rightarrow_T^* C[q(t'_s)] \Rightarrow_T^* q_f(t')$ for some $q_f \in Q_f$ and $t'_s, t' \in \mathcal{T}_\Delta$.
2. There exists a subset $U(T)$ of Q such that for every rule $C_l[q_1(x_1), \ldots, q_k(x_k)] \to q(t_r) \in \delta$,
   - if $q \in U(T)$ then $t_r = \perp$ and $q_i \in U(T)$ for each $i \in [k]$, and
   - if $q \notin U(T)$ then (1) $t_r \neq \perp$ and (2) for each $i \in [k]$, $q_i \in U(T)$ if and only if $x_i \notin \mathrm{var}(t_r)$.
3. If $q \in Q_f$ then q does not occur in the left-hand side of any rule in δ.

Note that for any $q \in U(T)$ and $t = Ct_s \in \mathrm{dom}(T)$ where $C \in \bar{\mathcal{C}}_\Sigma(\{x_*\})$, if $t \Rightarrow_T^* C[q(t'_2)]$ then $t'_2 = \perp$ and the final output for t does not contain $\perp$. That is, the intermediate output at q is always $\perp$ and it is eventually abandoned. Conversely, for $q \in Q - U(T)$, the intermediate output at q is in $\mathcal{T}_{\Delta - \{\perp\}}$ and it is contained in the final output. For every xbot T, a reduced xbot equivalent with T can be constructed in the same way as the construction for bots [9].

2.3 Determinacy and Subsumption of Tree Transducers

Let Π_1 and Π_2 be arbitrary classes of tree transducers.

Definition 1 (Determinacy). *Let T_1 and T_2 be tree transducers in Π_1 and Π_2, respectively, such that* $\mathrm{dom}(T_2) \subseteq \mathrm{dom}(T_1)$*. T_1* determines *T_2 iff there exists a partial function f such that $[\![T_2]\!] = f \circ [\![T_1]\!]$. Π_1 is called the* determiner class *and Π_2 is called the* determinee class*.*

Definition 2 (Subsumption). *Let T_1 and T_2 be tree transducers in Π_1 and Π_2, respectively, such that* $\mathrm{dom}(T_2) \subseteq \mathrm{dom}(T_1)$*. T_1* subsumes *T_2 with respect to Π_2 iff there exists a single-valued transducer $T_3 \in \Pi_2$ such that $[\![T_2]\!] = [\![T_3]\!] \circ [\![T_1]\!]$.*

From the definition, if T_1 subsumes T_2 then T_1 determines T_2. Conversely, even if there exists some function f such that $[\![T_2]\!] = f \circ [\![T_1]\!]$, f cannot always be induced by some transducer in Π_2 in general.

If determinacy is decidable for a determiner class Π_1 and a determinee class Π_2, we simply say determinacy is decidable for (Π_1, Π_2). We will use a similar notation for subsumption.

3 Determinacy

3.1 Decidability for (*sl*-xbots, *s*-bots)

We consider the problem of deciding whether, given single-valued linear xbot (*sl*-xbot) T_1 and single-valued bot (*s*-bot) T_2 such that $\mathrm{dom}(T_2) \subseteq \mathrm{dom}(T_1)$, T_1 determines T_2 or not, based on the next proposition.

Proposition 3. *For any two single-valued transducers T_1 and T_2 such that* $\mathrm{dom}(T_2) \subseteq \mathrm{dom}(T_1)$*, T_1 determines T_2 iff $[\![T_2]\!] \circ [\![T_1]\!]^{-1}$ is a partial function, where $[\![T_1]\!]^{-1} = \{(t', t) \mid (t, t') \in [\![T_1]\!]\}$.*

According to Proposition 3, given *sl*-xbot T_1 and *s*-bot T_2, our decision algorithm works as follows:

Step 1: Construct a transducer T_1^{inv} such that $[\![T_1^{inv}]\!] = [\![T_1]\!]^{-1}$;
Step 2: Construct a transducer T_3 such that $[\![T_3]\!] = [\![T_2]\!] \circ [\![T_1^{inv}]\!]$;
Step 3: Decide whether T_3 is single-valued.

In Step 1, the inverse transducer T_1^{inv} of T_1 is computed. T_1^{inv} is not necessarily an *l*-xbot. Due to this, we introduce a slightly larger class, linear extended bottom-up tree transducers with *grafting* (*l*-xbot$^{+\mathrm{g}}$ for short), that can represent not only inverses of *l*-xbots but also the composition of the inverses with *s*-bots. In Step 2, an xbot$^{+\mathrm{g}}$ T_3 which represents the composition of T_1^{inv} followed by T_2 is constructed. Lastly, it is determined whether the composition transducer T_3 is single-valued.

Before we explain the detail of each step, we give an example, which shows that even the inverse of an *sl*-bot cannot always be expressed by any *l*-xbot.

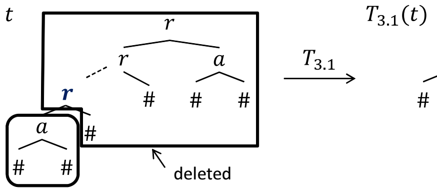


Fig. 2. A transducer $T_{3.1}$

Example 4. Let $\Sigma = \{r, a, \#\}$ and $\Delta = \{a, \#\}$. Consider an *sl*-bot $T_{3.1} = (\{q_r, q\}, \Sigma, \Delta, \{q_r\}, \delta)$ where

$$\delta = \{\ \# \to q(\#), \quad a(q(x_1), q(x_2)) \to q(a(x_1, x_2)),$$
$$r(q(x_1), q(x_2)) \to q_r(x_1), \quad r(q_r(x_1), q(x_2)) \to q_r(x_1)\}.$$

In Fig. 2, t is transformed by $T_{3.1}$, which leaves only the subtree at the left child of the bottom-most r-node. There is an infinite number of trees t' such that $T_{3.1}(t') = T_{3.1}(t)$ because the inverse of $T_{3.1}$ allows to insert any number of r-labeled ancestor nodes having arbitrary trees in $\mathcal{T}_{\Sigma-\{r\}}$ as their right subtrees. For any l-xbot T without ϵ-rules, the image of a tree t by T is finite. Even if ϵ-rules are allowed, no l-xbot allows to insert a node having an arbitrary tree in $\mathcal{T}_{\Sigma-\{r\}}$ as its right subtree. Therefore, there is no l-xbot T such that $[\![T]\!] = [\![T_{3.1}]\!]^{-1}$.

To express the inverse of $T_{3.1}$ in Example 1, a transducer has to, for an input tree, insert any number of internal nodes and subtrees non-deterministically. To capture the inverse of *sl*-xbots, we extend xbots by *grafting*. We denote a tree transducer in the class by an xbot$^{+\mathrm{g}}$ for short. A grafting is represented by a special variable $\langle L\rangle$, called a g-variable, where $L \subseteq \mathcal{T}_\Delta$. When $L = L(A)$ where A is a TA over Δ, we often write $\langle A\rangle$ instead of $\langle L(A)\rangle$. A g-variable can occur as a symbol of rank 0 in the right-hand side of a rule. Let $G(\Delta)$ be the set of all the g-variables $\langle L\rangle$ where $L \subseteq \mathcal{T}_\Delta$. Let $\tilde{\mathcal{T}}_\Delta(X_i)$ denote the set of trees over Δ with X_i and $G(\Delta)$. Note that for $\tilde{t} \in \tilde{\mathcal{T}}_\Delta(X_i)$, $\mathrm{var}(\tilde{t})$ does not contain any g-variable. For $\tilde{t} \in \tilde{\mathcal{T}}_\Delta(X_i)$, let $S(\tilde{t})$ be the set of trees in $\mathcal{T}_\Delta(X_i)$ obtained from $\tilde{t}$ by replacing each g-variable $\langle L\rangle$ with a tree in L. Formally, a transduction rule of an xbot$^{+\mathrm{g}}$ is the form $C_l[q_1(x_1), \ldots, q_k(x_k)] \to q(\tilde{t}_r)$ where $k \in \mathbb{N}$, $C_l \in \bar{\mathcal{C}}_\Sigma(X_k)$, $\tilde{t}_r \in \tilde{\mathcal{T}}_\Delta(X_k)$, and $q, q_1, \ldots, q_k$ are states. The move relation by a rule $C_l[q_1(x_1), \ldots, q_k(x_k)] \to q(\tilde{t}_r)$ is as follows: if $t|_p = C_l[q_1(t_1), \ldots, q_k(t_k)]$ where $t_1, \ldots, t_k \in \mathcal{T}_\Delta$, then $t \Rightarrow t[q(t_r[t_1, \ldots, t_k])]_p$ where $t_r \in S(\tilde{t}_r)$. For an xbot$^{+\mathrm{g}}$, we write an xbot$^{+\mathrm{g(R)}}$ when L is regular for each g-variable $\langle L\rangle$. Also, we write an xbot$^{+\mathrm{g(B(R))}}$ when each g-variable is in the form of $\langle T(A)\rangle$ for some bot T and TA A.

Example 5. Consider an l-xbot$^{+\mathrm{g(R)}}$ $T_{3.2} = (\{q, q_r\}, \Delta, \Sigma, \{q_r\}, \delta')$ where

$$\delta' = \{\ \# \to q(\#), \quad a(q(x_1), q(x_2)) \to q(a(x_1, x_2)),$$
$$q(x_1) \to q_r(r(x_1, \langle A\rangle)), q_r(x_1) \to q_r(r(x_1, \langle A\rangle))\}$$

and A is a TA such that $L(A) = \mathcal{T}_{\Sigma-\{r\}}$. Then, $T_{3.2}$ induces the inverse of $T_{3.1}$.

Steps 1 to 3 of the decision algorithm can be refined as follows.

Step 1: Inversion of *sl*-xbots. We provide a way to construct an l-xbot$^{+\mathrm{g}}$ representing the inverse of an sl-xbot. Intuitively, we just swap the input and output of each rules. However, we must take care of variables occurring only in the left-hand side, which mean deletions of subtrees. In swapping, g-variables are added instead of the variables.

Let $T = (Q, \Sigma, \Delta, Q_f, \delta)$ be an l-xbot. The swapping procedure is as follows.

1. Construct a TA $A_T = (Q, \Sigma, Q_f, \gamma)$ where $\gamma = \{(q, C_l[q_1, \ldots, q_k]) \mid C_l[q_1(x_1), \ldots, q_k(x_k)] \to q(C_r) \in \delta\}$. Note that $L(A_T) = \mathrm{dom}(T)$.
2. Construct an l-xbot$^{+\mathrm{g(R)}}$ $T' = (Q, \Delta, \Sigma, Q_f, \delta')$ such that δ' is the smallest set satisfying the following condition: Let $C_l[q_1(x_1), \ldots, q_k(x_k)] \to q(C_r)$ be an arbitrary rule in δ. Let θ_l be the substitution such that $\theta_l(x_i) = q_i(x_i)$ for each $i \in [k]$, θ_r be the substitution such that $\theta_r(x_i) = x_i$ if $x_i \in \mathrm{var}(C_r)$ and $\theta_r(x_i) = \langle A_T(q_i)\rangle$ otherwise. Moreover, let θ_n be the substitution for normalization, which is the bijective function from $\mathrm{var}(C_r)$ to $X_{k'}$ ($k' = |\mathrm{var}(C_r)|$) making $(C_r\theta_l)\theta_n$ normalized. Then, $(C_r\theta_l)\theta_n \to (C_l\theta_r)\theta_n \in \delta'$.

Lemma 6. *For any l-xbot T, an l-xbot$^{+\mathrm{g(R)}}$ T^{inv} such that $[\![T^{inv}]\!] = [\![T]\!]^{-1}$ can be constructed.*

Step 2: Composition of *l*-xbot$^{+\mathrm{g(R)}}$ and *s*-bot. This step constructs an xbot$^{+\mathrm{g}}$ equivalent with the composition of the l-xbot$^{+\mathrm{g(R)}}$ T_1^{inv} followed by an s-bot T_2.

Lemma 7. *For any l-xbot$^{+\mathrm{g(R)}}$ T and bot T', an xbot$^{+\mathrm{g(B(R))}}$ T'' such that $[\![T'']\!] = [\![T']\!] \circ [\![T]\!]$ can be constructed.*

Proof. The lemma can be shown in a similar way to the proof of the closure property of l-bots under the composition [3]. The difference is the existence of g-variables. Recall that a tree t in $L(A)$ is inserted at g-variable $\langle A\rangle$. On the composition transducer, we just insert the image of t by $T'(q)$ where q is the state at which T' processes t in the tree output by T. That is, we replace $\langle A\rangle$ with $\langle T'(q)(A)\rangle$. □

Step 3: Deciding single-valuedness of xbot$^{+\mathrm{g(B(R))}}$. This step decides whether the xbot$^{+\mathrm{g(B(R))}}$ obtained in Step 2 is single-valued. It is known that single-valuedness of bots is decidable in polynomial time [9]. However, the class of transformations induced by xbot$^{+\mathrm{g}}$s is a proper superclass of the class induced by bots.

Let T_3 be the xbot$^{+\mathrm{g(B(R))}}$ obtained in Step 2. The overview of Step 3 is as follows:

Step 3-1: Construct a reduced xbot $T_{3.1}$ equivalent with T_3 by eliminating g-variables. If there is no xbot equivalent with T_3, answer that T_3 is not single-valued and halt. Otherwise, go to 3-2.

Step 3-2: Construct a reduced xbot^{-e} $T_{3.2}$ equivalent with $T_{3.1}$. If there is no xbot^{-e} equivalent with $T_{3.1}$, answer that T_3 is not single-valued and halt. Otherwise, go to 3-3.
Step 3-3: Decide whether $T_{3.2}$ is single-valued or not.

We further refine the above sub-steps as follows.

Step 3-1*: Eliminating g-variables.* We show the following lemma for Step 3-1.

Lemma 8. *Let $T = (Q, \Sigma, \Delta, Q_f, \delta)$ be a reduced xbot$^{+\mathrm{g}}$. If T has a rule whose right-hand side has a state $q \in Q - U(T)$ and a g-variable $\langle L \rangle$ such that $|L| \geq 2$, then T is not single-valued.*

For a bot T, a TA A and any $k \in \mathbb{N}$, it can be checked whether $|T(A)| \geq k$. From the above lemma, Step 3-1 can be done as follows:

(i) For each rule with grafting $\langle T(A) \rangle$ of T_3,
  – if $T(A) = \emptyset$ then delete the rule, and
  – if $T(A) = \{t\}$ for some tree t then replace $\langle T(A) \rangle$ with t.
(ii) Construct an equivalent reduced xbot$^{+\mathrm{g(B(R))}}$ $T_{3.1}$.
(iii) If $T_{3.1}$ has a rule containing $\langle T(A) \rangle$ with $|T(A)| \geq 2$, answer that T_3 is not single-valued and halt.

If the condition at (iii) does not hold, $T_{3.1}$ is an xbot, without g-variables.

Step 3-2*: Eliminating ϵ-rules.* We show two lemmas before giving the procedure of Step 3-2. We will use an idea similar to the proof of Proposition 10 of [4].

We say that a nonempty subset δ_e of ϵ-rules is *repeatedly-producing at state q* if $q(x_*) \Rightarrow^*_{\delta_e} q(t)$ for some tree $t \in \bar{\mathcal{T}}_\Delta(\{x_*\}) - \{x_*\}$, where $\Rightarrow^*_{\delta_e}$ means zero ore more applications of rules in δ_e.

Lemma 9. *Let $T = (Q, \Sigma, \Delta, Q_f, \delta)$ be a reduced xbot. If there is a subset δ_e of ϵ-rules in δ repeatedly-producing at some $q \in Q - U(T)$, then T is not single-valued.*

After the fashion of the reference [4], we call a state $q \in Q$ an *end state* if there exists an input-consuming rule whose left-hand side has q. The set of all end states of Q is denoted by $E(T)$. For each input-consuming rule $\rho = (C_l[q_1(x_1), \ldots, q_k(x_k)] \to q(t_r)) \in \delta$, let $\mathrm{rhs}(\rho) = \{q'(t) \mid q(t_r) \Rightarrow^*_T q'(t), q' \in E(T) \cup Q_f\}$. Note that only ϵ-rules can be used in the derivation $q(t_r) \Rightarrow^*_T q'(t)$.

Lemma 10. *Let $T = (Q, \Sigma, \Delta, Q_f, \delta)$ be a reduced xbot. If there is no subset δ_e of ϵ-rules in δ repeatedly-producing at any $q \in Q - U(T)$, then* $\mathrm{rhs}(\rho)$ *is finite for every rule ρ of T and an xbot^{-e} equivalent with T can be constructed.*

According to Lemmas 9 and 10, Step 3-2 consists of the following two substeps:

(i) Construct the weighted graph $G_{rp} = (Q - U(T_{3.1}), E_e)$ from $T_{3.1} = (Q, \Sigma, \Delta, Q_f, \delta)$ where $E_e = \{(q, q') \mid q(x_1) \to q'(t) \in \delta, t \in \bar{\mathcal{T}}_\Delta(\{x_1\})\}$, and the weight of each (q, q') is 1 if there is a rule $q(x_1) \to q'(t)$ such that t includes at least one output symbol, and otherwise 0. Find a cycle whose weight is at least one. If such a cycle exists, answer that T_3 is not single-valued and halt.
(ii) Construct an equivalent reduced xbot^{-e} $T_{3.2}$.

Step 3-3: *Deciding single-valuedness of xbot*$^{-e}$. In this substep, it is decided whether $T_{3.2}$ is single-valued or not. The idea of the proof is the same as that of the proof of the decidability of k-valuedness of bottom-up tree transducers [10]. While the proof in [10] uses the Engelfriet's property, we use a variant of the property (Lemma 11) to prove the decidability of single-valuedness of xbot^{-e}s.

We give some notations for the property. Let $\mathcal{T}_\Sigma[X_n] = \bar{\mathcal{T}}_\Sigma(X_n) \cup \mathcal{T}_\Sigma$, that is, every $t \in \mathcal{T}_\Sigma[X_n]$ has all the variables in X_n or has no variable. For $t, s \in \mathcal{T}_\Sigma[X_n]$, ts is the tree obtained from t by replacing each variable with s. Note that $ts = t$ if t has no variable. For $m \in [n]$, let $\mathcal{T}_\Sigma^{n,m}[X_n] = \mathcal{T}_\Sigma^{m-1} \times \mathcal{T}_\Sigma[X_n] \times \mathcal{T}_\Sigma^{n-m}$. For $\mathbf{t} \in \mathcal{T}_\Sigma^{n,m}[X_n]$, we denote by $t^{(i)}$ the ith element of $\mathbf{t}$, i.e., $\mathbf{t} = (t^{(1)}, \dots, t^{(n)})$. For $s \in \mathcal{T}_\Sigma[X_n]$ and $\mathbf{t} \in \mathcal{T}_\Sigma^{n,m}[X_n]$, $s\mathbf{t}$ is the tree obtained from s by replacing x_i with $t^{(i)}$ for all $i \in [n]$. Let $\mathbf{tu} = (t^{(1)}\mathbf{u}, \dots, t^{(n)}\mathbf{u})$ for $\mathbf{u} \in \mathcal{T}_\Sigma^{n,m}[X_n]$. Notice that since $\mathbf{t} \in \mathcal{T}_\Sigma^{n,m}[X_n]$, so is $\mathbf{tu}$. For $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3, \mathbf{t}_4, \mathbf{t}_5 \in \mathcal{T}_\Sigma^{n,m}[X_n]$ and $S = \{i_1, \dots, i_{|S|}\} \subseteq [1,5]$, let $\mathbf{t}_S = \mathbf{t}_{i_1} \cdots \mathbf{t}_{i_{|S|}}$ where $i_j < i_{j+1}$ for $j \in [|S|-1]$.

Now, we give a variant of Engelfriet's Property for tuples of trees.

Lemma 11. *Let n, n' be arbitrary positive integers, and $m \in [n], m' \in [n']$. Suppose that $t_0 \in \mathcal{T}_\Sigma[X_n]$, $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3, \mathbf{t}_4, \mathbf{t}_5 \in \mathcal{T}_\Sigma^{n,m}[X_n]$, $t'_0 \in \mathcal{T}_\Sigma[X_{n'}]$, $\mathbf{t}'_1, \mathbf{t}'_2, \mathbf{t}'_3, \mathbf{t}'_4, \mathbf{t}'_5 \in \mathcal{T}_\Sigma^{n',m'}[X_{n'}]$. If $t_0\mathbf{t}_S = t'_0\mathbf{t}'_S$ for every S s.t. $\{5\} \subseteq S \subset [1,5]$, then $t_0\mathbf{t}_{[1,5]} = t'_0\mathbf{t}'_{[1,5]}$.*

Next, in order to argue in a similar way to the proof of Theorem 2.2(i) in the reference [10], we decompose the left-hand side of each rule into several rules each of which has only one input symbol. Actually, we construct a *multi bottom-up tree transducer* (mbot) [4] equivalent with a given xbot^{-e}. An mbot is a bot whose states might have ranks different from one. Intuitively, we decompose each rule ρ of the xbot^{-e} by adding a state for each intermediate position of the left-hand side tree l of ρ. The added states might have rank different from one to maintain two or more intermediate output trees until the obtained mbot reaches the state corresponding to the root of l.

Example 12. Assume that an xbot^{-e} T contains the transduction rule $\rho = a(b(q_1(x_1), q_2(x_2), q_3(x_3)), q_4(x_4)) \to q(c(x_1, x_2, x_4))$. Then, the mbot T_a obtained by decomposing T contains the rules $b(q_1(x_1), q_2(x_2), q_3(x_3)) \to q_1^\rho(x_1, x_2)$ and $a(q_1^\rho(x_1, x_2), q_4(x_4)) \to q(c(x_1, x_2, x_4))$ (See Fig. 3). Note that q_1^ρ maintains x_1 and x_2 but not x_3 because x_3 does not occur in the right-hand side of ρ.

Lemma 13. *Let $T_a = (Q_a, \Sigma, \Delta, Q_f, \delta_a)$ be the mbot obtained from an xbot^{-e} $T = (Q, \Sigma, \Delta, Q_f, \delta)$ by the above decomposition. Then, for every $q \in Q_a$ and $C \in \bar{\mathcal{C}}_\Sigma(\{x_*\})$, if $C[q(x_1, \dots, x_k)] \Rightarrow_{T_a}^+ q(t_1, \dots, t_k)$, then $(t_1, \dots, t_k) \in \mathcal{T}_\Delta^{k,m}[X_k]$ for some $m \in [k]$.*

Henceforth, we denote $q(t_1, \dots, t_k)$ by $q(\mathbf{t})$ where $\mathbf{t} = (t_1, \dots, t_k)$.

Lemma 14. *Let $T_a = (Q_a, \Sigma, \Delta, Q_f, \delta_a)$ be the mbot obtained from an xbot^{-e} T by the above decomposition. Assume that T_a has n states and the maximum arity of states is k_m. T_a is not single-valued if and only if there is a tree t of depth less than $5 \cdot (n \cdot k_m)^2$ such that $|[\![T_a]\!](t)| > 1$.*

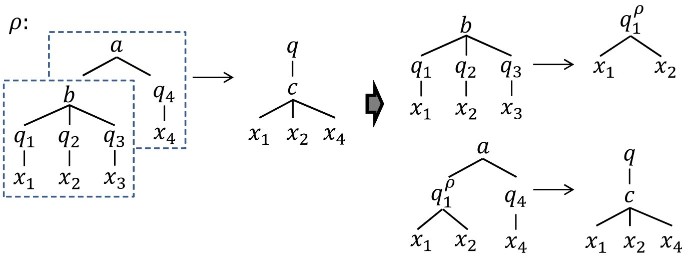


Fig. 3. An example of decomposing an xbot^{-e} to an mbot

Proof. The if part is trivial and so we prove the only if part. Assume that $t \in \mathcal{T}_\Sigma$ is a tree of minimal size such that there are two distinct derivations $t \Rightarrow^*_{T_a} q_{f1}(t_{o1})$ and $t \Rightarrow^*_{T_a} q_{f2}(t_{o2})$ where $q_{f1}, q_{f2} \in Q_f$, and $t_{o1} \neq t_{o2}$. For a contradiction, assume that the depth of t is greater than or equal to $5 \cdot (n \cdot k_m)^2$. Then, by Lemma 13, there are two states $q_1, q_2 \in Q_a$ with ranks n_1 and n_2 respectively, $C_j \in \bar{\mathcal{C}}_\Sigma(\{x_*\})$ $(0 \leq j \leq 4)$, $C_5 \in \mathcal{T}_\Sigma$, and for $i \in \{1,2\}$, $m_i \in [n_i]$, $t^i_0 \in \mathcal{T}_\Delta[X_{n_i}]$, and $\mathbf{t}^i_j \in \mathcal{T}^{n_i,m_i}_\Delta[X_{n_i}]$ $(j \in [5])$ such that

$$\begin{aligned} t = C_0C_1C_2C_3C_4C_5 &\Rightarrow^*_{T_a} C_0C_{[1,4]}[q_i(\mathbf{t}^i_5)] \Rightarrow^*_{T_a} C_0C_{[1,3]}[q_i(\mathbf{t}^i_{[4,5]})] \\ &\Rightarrow^*_{T_a} \cdots \Rightarrow^*_{T_a} C_0[q_i(\mathbf{t}^i_{[1,5]})] \Rightarrow^*_{T_a} q_{fi}(t^i_0\mathbf{t}^i_{[1,5]}). \end{aligned}$$

By the minimality of t, we have $t^1_0\mathbf{t}^1_S = t^2_0\mathbf{t}^2_S \in [\![T_a]\!](C_0C_S)$ for every S s.t. $\{5\} \subseteq S \subset [1,5]$. From Lemma 11, $t_{o1} = t^1_0\mathbf{t}^1_{[1,5]} = t^2_0\mathbf{t}^2_{[1,5]} = t_{o2}$. This is a contradiction. □

Theorem 15. *It is decidable whether a given xbot^{-e} is single-valued. It is also decidable whether a given xbot$^{+\mathrm{g(B(R))}}$ is single-valued.*

Theorem 16. *Determinacy is decidable for (sl-xbots, s-bots).*

3.2 Undecidability for Other Classes

We show that determinacy is undecidable for (non-linear) s-bots as the determiner class. We prove the undecidability of determinacy for homomorphism tree transducers (homs) [3], which is a proper subclass of not only s-bots but also single-valued top-down tree transducers (s-top). Let *id* be the class of the identity transductions on $\mathcal{T}_\Sigma$ for any alphabet Σ.

Theorem 17. *Determinacy is undecidable for (homs, id).*

Proof. It can be shown by reduction from injectivity of homs, which is known to be undecidable [7]. Consider an arbitrary hom T. Then, it holds that $[\![T]\!]$ is injective if and only if T determines the identity transducer T_{id}. □

Corollary 18. *Determinacy is undecidable for (s-bots, id) and (s-tops, id).*

Moreover, we have the undecidability result for deterministic monadic second-order logic defined tee transducers (dmsotts) [2,5], which is a proper superclass of *sl*-xbots.

Theorem 19. *Determinacy is undecidable for (dmsotts, id_R) where id_R is the class of the identities whose domains are regular tree languages.*

Proof. It can be shown by reduction from ambiguity of context-free grammars. Consider an arbitrary context-free grammar G. Then, there is a dmsott T_G which transforms any derivation tree of each string $s \in L(G)$ to s. Thus, G is ambiguous if and only if T_G determines the identity transducer T_{id} such that $\mathrm{dom}(T_{id}) = \mathrm{dom}(T_G)$. □

4 Subsumption

We show that subsumption is decidable for (*sl*-xbots, *s*-bots). As shown in Section 3, given an *sl*-xbot T_1 and an *s*-bot T_2, if T_1 determines T_2, we can construct a reduced *s*-xbot^{-e} T_3 such that $[\![T_3]\!] = [\![T_2]\!] \circ [\![T_1]\!]^{-1}$. So, in order to decide subsumption, we should decide whether there is a bot equivalent with T_3. The next lemma provides a necessary and sufficient condition for an *s*-xbot^{-e} to have an equivalent bot.

Lemma 20. *Let $T = (Q, \Sigma, \Delta, Q_f, \delta)$ be a reduced s-xbot^{-e}. An s-bot equivalent with T can be constructed iff* (X) *for every rule $C_l[q_1(x_1), \dots, q_k(x_k)] \to q(t_r) \in \delta$ and any three variables x_{i_1}, x_{i_2}, $x_{i_3} \in \mathrm{var}(t_r)$, if*

(X1) $\mathrm{rng}(T(q_{i_j}))$ *is infinite for all $j \in [3]$, and*
(X2) $\mathrm{nca}(p_1, p_2) \succ \mathrm{nca}(p_1, p_3)$ *where $\{p_j\} = pos_{x_{i_j}}(C_l)$ for $j \in [3]$, then*
(X3) *the minimal suffix $t_s \in \mathcal{T}_\Sigma(X_k)$ such that $t_r = t_p t_s$ for some $t_p \in \bar{\mathcal{T}}_{\Sigma \cup X_k - \{x_{i_1}, x_{i_2}\}}(\{x_*\})$ does not contain x_{i_3}.*

Proof Sketch. Assume (X) does not hold and we can construct an *s*-bot T' equivalent with a given *s*-xbot^{-e} T. Since (X) does not hold, there is a rule $C_l[q_1(x_1), \dots, q_k(x_k)] \to q(t_r) \in \delta$ and x_{i_1}, x_{i_2}, $x_{i_3} \in \mathrm{var}(t_r)$ such that (X1) and (X2) hold but (X3) does not. Let $p_{12} = \mathrm{nca}(p_1, p_2)$ in (X2), and t_s be the minimal suffix of t_r in (X3). Since T' is an *s*-bot equivalent with T, T' must have rules of which left-hand sides 'cover' the subtree $C_l|_{p_{12}}$, which contains x_{i_1} and x_{i_2} and does not contain x_{i_3}. Also, since $C_l[x_*]_{p_{12}}$ does not contain x_{i_1} and x_{i_2}, some suffix t'_s of t_r in the right-hand side such that $t_r = t'_p t'_s$ for some $t'_p \in \bar{\mathcal{T}}_{\Sigma \cup X_k - \{x_{i_1}, x_{i_2}\}}(\{x_*\})$ should be generated by T' corresponding to $C_l|_{p_{12}}$. However, the minimal suffix t_s contains x_{i_3}, and thus so does t'_s. That is, t'_s including x_{i_3} should be generated from $C_l|_{p_{12}}$ without x_{i_3}, which leads a contradiction. Conversely, if (X) holds, we can divide each rule of T into non-extended rules, each of which has exactly one symbol in the left-hand side. □

For any xbot T, it can be decided whether $\mathrm{rng}(T)$ is infinite. Thus, it is decidable whether there is an *s*-bot equivalent with a given *s*-xbot^{-e}.

Theorem 21. *Subsumption is decidable for (sl-xbots, s-bots).*

5 Conclusion

We have shown that determinacy and subsumption are decidable for single-valued linear extended bottom-up tree transducers as the determiner class and single-valued bottom-up tree transducers as the determinee class. As for more powerful classes, we have shown that determinacy is undecidable for single-valued top-down/bottom-up tree transducers (s-tops/bots) and deterministic MSO tree transducers (dmsotts) as the determiner class.

As future work, we will investigate whether subsumption for more powerful classes, such as s-tops/bots and dmsotts, is decidable or not. Though determinacy is undecidable for s-tops/bots and dmsotts, decidability of subsumption for the classes is still open. We also consider whether, given two transducers T_1 and T_2 in the classes such that T_1 subsumes T_2, a transducer T_3 such that $[\![T_2]\!] = [\![T_3]\!] \circ [\![T_1]\!]$ can be effectively constructed or not.

Acknowledgments. This research is supported in part by Grant-in-Aid for Scientific Research (B) 23300008 and Grant-in-Aid for Young Scientists (B) 22700099 from Japan Society for the Promotion of Science.

References

1. Afrati, F.N.: Determinacy and query rewriting for conjunctive queries and views. Theoretical Computer Science 412(11), 1005–1021 (2011)
2. Courcelle, B.: Monadic second-order definable graph transductions: A survey. Theoretical Computer Science 126(1), 53–75 (1994)
3. Engelfriet, J.: Bottom-up and top-down tree transformations - a comparison. Mathematical Systems Theory 9(3), 198–231 (1975)
4. Engelfriet, J., Lilin, E., Maletti, A.: Extended multi bottom-up tree transducers. Acta Informatica 46, 561–590 (2009)
5. Engelfriet, J., Maneth, S.: The equivalence problem for deterministic MSO tree transducers is decidable. Information Processing Letters 100(5), 206–212 (2006)
6. Fan, W., Geerts, F., Zheng, L.: View determinacy for preserving selected information in data transformations. Information Systems 37(1), 1–12 (2012)
7. Fulop, Z., Gyenizse, P.: On injectivity of deterministic top-down tree transducers. Information Processing Letters 48(4), 183–188 (1993)
8. Hashimoto, K., Sawada, R., Ishihara, Y., Seki, H., Fujiwara, T.: Determinacy and subsumption for single-valued bottom-up tree transducers. Tech. rep., NAIST (2012), `http://isw3.naist.jp/IS/TechReport/report/2012002.pdf`
9. Seidl, H.: Single-valuedness of tree transducers is decidable in polynomial time. Theoretical Computer Science 106(1), 135–181 (1992)
10. Seidl, H.: Equivalence of finite-valued tree transducers is decidable. Theory of Computing Systems 27, 285–346 (1994)
11. Zheng, L., Chen, H.: Determinacy and rewriting of conjunctive queries over unary database schemas. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 1039–1044 (2011)

Revealing vs. Concealing: More Simulation Games for Büchi Inclusion

Milka Hutagalung, Martin Lange, and Etienne Lozes

School of Electr. Eng. and Computer Science, University of Kassel, Germany[*]

Abstract. We address the problem of deciding language inclusion between two non-deterministic Büchi automata. It is known to be PSPACE-complete and finding techniques that are efficient in practice is still a challenging problem. We introduce two new sequences of simulation relations, called multi-letter simulations, in which Verifier has to reproduce Refuter's moves taking advantage of a forecast. We compare these with the multi-pebble games introduced by Etessami. We show that multi-letter simulations, despite being more restrictive than multi-pebble ones, have a greater potential for an incremental inclusion test, for their size grows generally slower. We evaluate this idea experimentally and show that incremental inclusion testing may outperform the most advanced Ramsey-based algorithms by two orders of magnitude.

1 Introduction

Nondeterministic Büchi automata (NBA) are an important formalism and defacto standard for the specification and verification of reactive systems. In the automata-theoretic approach to formal verification [18], problems about programs get translated into decision problems on automata, for instance NBA. Satisfiability of a specification corresponds to language non-emptiness which is NLOGSPACE-complete for NBA. Hence, it boils down to some reachability questions which can be solved efficiently.

The step from wanted to unwanted (or vice-versa) program behaviour corresponds to the complementation problem for NBA. It is known that this is inherently exponential. This explains why other decision problems on NBA are harder than emptiness, for example inclusion—corresponding to the question of whether one specification supersedes another—is PSPACE-complete. Note that $L(\mathcal{A}) \subseteq L(\mathcal{B})$ iff $L(\mathcal{A}) \cap \overline{L(\mathcal{B})} = \emptyset$. It is possible to construct an NBA recognising $L(\mathcal{A}) \cap \overline{L(\mathcal{B})}$ which is in general exponentially larger than $\mathcal{B}$ and which can then be tested for emptiness. This results in an NPSPACE procedure, and Savitch's Theorem [15] brings it down to PSPACE.

A large amount of work in the area of automata theory for verification is devoted to avoiding explicit complementation because none of the existing complementation procedures [14,17,12] is widely accepted to be good enough for

[*] The European Research Council has provided financial support under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 259267.

practical purposes. Regarding the inclusion problem, there is for instance the so-called Ramsey-based approach [7,1,2] which essentially uses the computational content of Büchi's original proof of complementability of NBA. It clearly does not avoid PSPACE-hardness but it can sometimes outperform algorithms based on explicit complementation.

Since the early works of Dill et al [4], there has also been an approach based on simulation relations. The notion of simulation that reflects the Büchi acceptance condition, called *fair simulation* [10], can be described in terms of a game between two players, Refuter and Verifier. Consider two NBA $\mathcal{A}$ and $\mathcal{B}$ over some alphabet Σ. The game proceeds for possibly infinitely many rounds, starting with two pebbles being placed on the initial states of $\mathcal{A}$ and $\mathcal{B}$. In each round, Refuter chooses some $a \in \Sigma$ and moves the $\mathcal{A}$-pebble along some a-transition. Verifier responds by moving the $\mathcal{B}$-pebble along some a-transition, too. Refuter wins if the infinite sequence of states pebbled in $\mathcal{A}$ forms an accepting run, and the one in $\mathcal{B}$ does not. A player loses as soon as he/she cannot move anymore.

There is a relatively simple encoding of this game as a parity game with 3 priorities. Thus, the winner in this game can be decided even in polynomial time. This game entails language inclusion, in the sense that language inclusion holds whenever Verifier has a winning strategy, but without surprise, the converse does not hold in general (otherwise we would have PSPACE=PTIME!). It is therefore reasonable to ask whether the fair simulation games can be refined in order to obtain something that is "closer" to language inclusion. The answer is yes, and it is helpful to consult some simple game-theoretic concepts for its explanation.

Fair simulation games are games of *perfect information*: at each time both players have full knowledge about the state of the game. Language inclusion, on the other hand, can be seen as a game of *imperfect information*, where refuter cannot observe Verifier's pebble, and Verifier can always revise his choice provided it remains consistent with all previous rounds. The anti-chain approach to language inclusion [5] can be seen as solving such a game, and, like the Ramsey-based approach, it is in general exponential but can outperform methods based on explicit complementation.

This reading of the problem suggests that an approximation of language inclusion based on fair simulation can be obtained by introducing some form of "opacity" in Verifier's moves. Since an approximation is either complete or it is not, it is useful to stratify the approximation based on some parameter $k \in \mathbb{N}$, such that smaller values of k, raising simpler games, can be tested first, yielding an incremental algorithm for deciding language inclusion. Such a parameterized opacity is obtained for the so-called k-pebble game [6]. This essentially gives Verifier a limited amount of resources in order to conceal information from Refuter. He now has up to k pebbles that he moves along a-transitions in $\mathcal{B}$ in order to respond to Refuter's choice a. The fair simulation game becomes the same as the 1-pebble fair simulation game, and with growing k, the k-pebble game gets "closer" to language inclusion.

In this paper, we introduce another family of approximations for language inclusion. That problem can be characterised by a simple one-round simulation game: first Refuter chooses an infinite word $w \in L(\mathcal{A})$, then Verifier produces an accepting run for it in $\mathcal{B}$. This is not a game of imperfect information anymore but it can be seen as a game with *infinite forecast*. A natural finite approximation of this is to let Verifier have a finite forecast on Refuter's moves. We call the result the *k-letter game*. It works similar to the ordinary fair simulation game but requires Refuter to always choose the next k letters of the infinite word to be constructed. This makes Refuter *reveal* more information about his subsequent choices to Verifier, whereas pebble games permit Verifier to *conceal* information about his past choices.

The k-letter games provide a new approach to the NBA inclusion problem. They can be used in order to devise an incremental language inclusion test. It successively solves games for increasing parameters k. Each iteration – which fixes k – can be done in polynomial time. The complexity grows exponentially with k but the base of this exponential is only the fixed (and often small) size of the underlying alphabet whereas, in the case of the k-pebble games, the base is the variable and usually not so small size of one of the input automata.

The rest of the paper is organised as follows. Sect. 2 recalls the necessary background about NBA and simulation games. Sect. 3 defines and examines so-called *static multi-letters simulations*. Sect. 4 refines them to *dynamic multi-letters simulations* which are theoretically better suited for approximating language inclusion. Sect. 5 reports on experiments that compare incremental inclusion tests based on multi-letter and multi-pebble games with one of the most advanced version of the Ramsey-based inclusion test.

2 Background

Words and Büchi Automata. As usual, we use Σ to denote a finite *alphabet* with $|\Sigma| \geq 2$, ε for the empty word, $\Sigma^*/\Sigma^+/\Sigma^\omega$ for the sets of all finite / non-empty finite / infinite words over Σ. For some $k \in \mathbb{N}$, Σ^k (resp $\Sigma^{\leq k}$) denotes the set of all words of length exactly (resp. at most) k. In the following, all language-theoretic concept are to be understood with respect to some fixed alphabet.

A non-deterministic Büchi automaton (NBA) is a tuple $\mathcal{A} = (Q, q_0, \Delta, F)$ where Q is a finite set of states with q_0 being a designated starting state, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and $F \subseteq Q$ is the set of accepting states. The *size* of $\mathcal{A}$ is measured in terms of its state space: $|\mathcal{A}| := |Q|$.

A *run* of $\mathcal{A}$ on a word $w = a_0 a_1 \cdots \in \Sigma^\omega$ is an infinite sequence $\rho = q_0, q_1, \ldots$ such that $(q_i, a_i, q_{i+1}) \in \Delta$ for all $i \geq 0$. Let $\mathit{inf}(q_0, q_1, \ldots) = \{q \mid$ there are infinitely many i with $q = q_i\}$. When using this notation we implicitly assume that $q_0, q_1, \ldots$ is an infinite sequence. The run is *accepting* if $\mathit{inf}(q_0, q_1, \ldots) \cap F \neq \emptyset$. The language of $\mathcal{A}$ is the set $L(\mathcal{A})$ of infinite words for which there exists an accepting run.

We write $q \xrightarrow{a} q'$ when $(q, a, q') \in \Delta$ assuming that the underlying NBA can be inferred from the context.

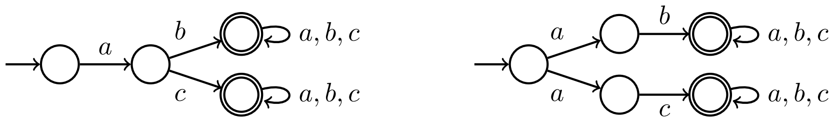
The *language inclusion* problem for NBA is the following. Given two NBA $\mathcal{A}$ and $\mathcal{B}$, decide whether or not $L(\mathcal{A}) \subseteq L(\mathcal{B})$ holds. We simply write $\mathcal{A} \sqsubseteq_{\mathsf{incl}} \mathcal{B}$ in that case.

Fair Simulation. The *fair simulation game* $\mathcal{G}_{\mathsf{fair}}$ [10] is played between Refuter and Verifier on two NBA $\mathcal{A} = (Q_{\mathcal{A}}, q^{\mathcal{A}}_{\mathsf{init}}, \Delta_{\mathcal{A}}, F_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, q^{\mathcal{B}}_{\mathsf{init}}, \Delta_{\mathcal{B}}, F_{\mathcal{B}})$, both controlling a pebble each, according to the following description.

Fair simulation game $\mathcal{G}_{\mathsf{fair}}(\mathcal{A}, \mathcal{B})$	
start:	$q_0 := q^{\mathcal{A}}_{\mathsf{init}}$, $q'_0 := q^{\mathcal{B}}_{\mathsf{init}}$
round i: $(i = 0, 1, \ldots)$	(1) Refuter chooses some $a \in \Sigma$ (2) Refuter chooses q_{i+1} such that $q_i \xrightarrow{a} q_{i+1}$ (3) Verifier chooses q'_{i+1} such that $q'_i \xrightarrow{a} q'_{i+1}$
winner:	Refuter wins if ... • Verifier cannot move anymore • $\mathit{inf}(q_0, q_1, \ldots) \cap F_{\mathcal{A}} \neq \emptyset$ and $\mathit{inf}(q'_0, q'_1, \ldots) \cap F_{\mathcal{B}} = \emptyset$ Verifier wins if ... • Refuter cannot move anymore • $\mathit{inf}(q_0, q_1, \ldots) \cap F_{\mathcal{A}} = \emptyset$ or $\mathit{inf}(q'_0, q'_1, \ldots) \cap F_{\mathcal{B}} \neq \emptyset$

We write $\mathcal{A} \sqsubseteq_{\mathsf{fair}} \mathcal{B}$ if Verifier has a winning strategy for $\mathcal{G}_{\mathsf{fair}}(\mathcal{A}, \mathcal{B})$. It is well-known that fair simulation approximates language inclusion in the sense that $\mathcal{A} \sqsubseteq_{\mathsf{fair}} \mathcal{B}$ implies $\mathcal{A} \sqsubseteq_{\mathsf{incl}} \mathcal{B}$ but not vice-versa. A counterexample for the converse implication is the following.

Example 1. Consider the following two NBA $\mathcal{A}$ (left) and $\mathcal{B}$ (right) over the alphabet $\Sigma = \{a, b, c\}$.



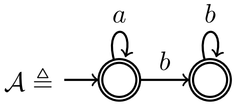
Clearly, we have $\mathcal{A} \sqsubseteq_{\mathsf{incl}} \mathcal{B}$. On the other hand, Refuter can win $\mathcal{G}_{\mathsf{fair}}(\mathcal{A}, \mathcal{B})$ because Verifier has to commit in the first round to some a-successor, and then Refuter can choose a b- or a c-successor whereas Verifier only has one of these choices.

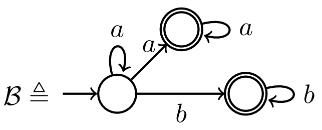
The game $\mathcal{G}_{\mathsf{fair}}(\mathcal{A}, \mathcal{B})$ can be reduced to a parity game [9] on a graph of size $\mathcal{O}(|\mathcal{A}| \cdot |\mathcal{B}| \cdot |\Sigma|)$. Thus, the winner of a fair simulation game is decidable. In the vertices, we keep track of the current state of $\mathcal{A}$, $\mathcal{B}$, and the letter a chosen by Refuter. The vertex reached by a move $q'_j \xrightarrow{a} q'_{j+1}$ of Verifier has a priority 2 if q'_{j+1} is a final state, and the vertex reached by a move $q_j \xrightarrow{a} q_{j+1}$ of Refuter has a priority 1 if q_{j+1} is a final state. Other vertices have priority 0. It is not too hard to check that player 0 wins the parity game iff Verifier has a winning strategy for $\mathcal{G}_{\mathsf{fair}}(\mathcal{A},\mathcal{B})$. In fact, winning strategies in these two games can easily be derived from one another.

Multi-Pebble Games. The *k-pebble game* for some $k \geq 1$ is a refinement of the fair simulation game [6]. Here we only give a brief sketch of its definition and state some important properties. A detailed definition is not necessary in order to follow the rest of this paper. The interested reader is referred to the literature for that purpose [6,3].

In the k-pebble game $\mathcal{G}^k_{\mathsf{peb}}(\mathcal{A},\mathcal{B})$ on two NBA $\mathcal{A}$ and $\mathcal{B}$, Refuter controls and moves a pebble on $\mathcal{A}$ as is done in the fair simulation game. Verifier now controls k pebbles on $\mathcal{B}$. In response to a letter a chosen by Refuter he chooses k (not necessarily different) a-successors of the currently k pebbled states in $\mathcal{B}$ and moves the pebbles there. The winning conditions for finite plays are the same as above. An infinite play is won by Verifier if the constructed infinite run in $\mathcal{A}$ is not accepting or if it can be guaranteed in some way that the pebbling in $\mathcal{B}$ "contained an accepting run". As an example, $\mathcal{A} \sqsubseteq^2_{\mathsf{peb}} \mathcal{B}$ holds for the two NBA of Ex. 1: in the first round, Verifier pushes a pebble in each of the two branches, and in the second round, he drops the irrelevant one, and continues like in the standard fair simulation game. It is hard to determine whether a sequence of pebblings contains an accepting run, roughly for the same reasons that make the power set construction unsuitable for determinising Büchi automata. However, since multi-pebble games only approximate language inclusion, it is possible to relax the winning condition even further and check whether the sequence of pebblings contains a run that is accepting w.r.t. a co-Büchi condition or, equivalently, whether all runs contained in it are accepting w.r.t. the Büchi condition. The determinisation problem for co-Büchi automata is conceptually simpler [13], and using this together with the possibility to drop pebbles yields a reasonable approximation and a direct reduction to finite parity games.

Example 2. For every $k \geq 1$, the k-pebble game for the two automata





is won by Refuter. He wins by always playing a: Verifier must keep a pebble on the first state of $\mathcal{B}$ to be ready for a possible b, i.e. he must not drop it at any time. Since this pebble does not visit an accepting state infinitely often, Refuter wins the game. Note that $L(\mathcal{A}) = L(\mathcal{B})$ nonetheless.

We write $\mathcal{A} \sqsubseteq^k_{\mathsf{peb}} \mathcal{B}$ if Verifier has a winning strategy for $\mathcal{G}^k_{\mathsf{peb}}(\mathcal{A},\mathcal{B})$. Obviously, we have $\sqsubseteq^1_{\mathsf{peb}} = \sqsubseteq_{\mathsf{fair}}$.

Proposition 3 ([6]). *For every $k \geq 1$ and NBA $\mathcal{A},\mathcal{B}$ over Σ there is a parity game of size at most $|\mathcal{A}| \cdot (2 \cdot |\mathcal{B}| + 1)^k \cdot (|\Sigma| + 1)$ and index 3 that is won by player 0 iff $\mathcal{A} \sqsubseteq^k_{\mathsf{peb}} \mathcal{B}$.*

Furthermore, these games give rise to the following hierarchy

$$\sqsubseteq^1_{\mathsf{peb}} \subsetneq \sqsubseteq^2_{\mathsf{peb}} \subsetneq \sqsubseteq^3_{\mathsf{peb}} \subsetneq \ \cdots \ \subsetneq \bigcup_{k \geq 1} \sqsubseteq^k_{\mathsf{peb}} \subsetneq \sqsubseteq_{\mathsf{incl}} \ . \tag{1}$$

3 The Static Multi-letter Games

We consider a new parametrised simulation game. In this game, Verifier moves only one pebble, but benefits from Refuter being forced to reveal more information to him/her in a single round.

Definition. Let $k \geq 1$. The *static k-letter game* is played between Refuter and Verifier on two NBA $\mathcal{A} = (Q_\mathcal{A}, q^\mathcal{A}_{\mathsf{init}}, \Delta_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, q^\mathcal{B}_{\mathsf{init}}, \Delta_\mathcal{B}, F_\mathcal{B})$, similar to the fair simulation game. However, in each round both players advance by k moves through the NBA instead of just 1 as it is done in the fair simulation game.

Static k-letters game $\mathcal{G}^k_{\mathsf{stat}}(\mathcal{A}, \mathcal{B})$	
start:	$q_0 := q^\mathcal{A}_{\mathsf{init}}$, $q'_0 := q^\mathcal{B}_{\mathsf{init}}$, $j := 0$
round i:	(1) Refuter chooses some $w = a_1 \dots a_k \in \Sigma^k$ (2) Refuter chooses $q_{j+1}, \dots, q_{j+k}$ such that $q_j \xrightarrow{a_1} q_{j+1} \xrightarrow{a_2} q_{j+2} \xrightarrow{a_3} \dots \xrightarrow{a_k} q_{j+k}$ (3) Verifier chooses $q'_{j+1}, \dots, q'_{j+k}$ such that $q'_j \xrightarrow{a_1} q'_{j+1} \xrightarrow{a_2} q'_{j+2} \xrightarrow{a_3} \dots \xrightarrow{a_k} q'_{j+k}$ (4) $j := j + k$
winner:	same as in fair simulation game

We write $\mathcal{A} \sqsubseteq^k_{\mathsf{stat}} \mathcal{B}$ if Verifier has a winning strategy for $\mathcal{G}^k_{\mathsf{stat}}(\mathcal{A}, \mathcal{B})$.

The game $\mathcal{G}^k_{\mathsf{stat}}(\mathcal{A}, \mathcal{B})$ can be modeled by a parity game on a graph of size $\mathcal{O}(|\mathcal{A}| \cdot |\mathcal{B}| \cdot (|\Sigma|^k + 1))$. In the vertices, we keep track the current state of $\mathcal{A}$, $\mathcal{B}$, and the word $w = a_1 \dots a_k$ chosen by Refuter. The vertex reached by a move $q'_j \xrightarrow{w} q'_{j+k}$ of Verifier has a priority 2 if a final state is seen along the move. The vertex reached by a move $q_j \xrightarrow{w} q_{j+k}$ of Refuter has priority 1 if a final state is seen along the move. Other vertices have priority 0. We can use a suitable result on parity games of index 3 [11], and obtain the following theorem.

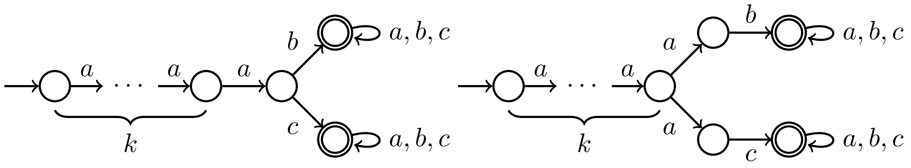
Theorem 4. *For every $k \geq 1$ and NBA $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{G}^k_{\mathsf{stat}}(\mathcal{A}, \mathcal{B})$ is decidable in time $\mathcal{O}((|\mathcal{A}| \cdot |\mathcal{B}| \cdot (|\Sigma|^k + 1))^3)$.*

Properties. It is quite obvious to see that the 1-letter static game is the same as the fair simulation game. Thus we have $\sqsubseteq^1_{\mathsf{stat}} = \sqsubseteq_{\mathsf{fair}}$. Furthermore, the static games approximate language inclusion in the following sense.

Theorem 5. *For every $k \geq 1$ and all NBA $\mathcal{A}$, $\mathcal{B}$ we have: $\mathcal{A} \sqsubseteq^k_{\mathsf{stat}} \mathcal{B}$ implies $\mathcal{A} \sqsubseteq_{\mathsf{incl}} \mathcal{B}$.*

Thus, static multi-letter games look like a good alternative to the multi-pebble games for incrementally searching for a proof of language inclusion. Indeed, the size of these games, while still growing exponentially, grows "only" as $\mathcal{O}(|\Sigma|^k)$, hence much slower than the multi-pebble games for typical inputs in which $\Sigma \ll |\mathcal{B}|$ (see Prop. 3).

However, the static multi-letter games do not form a hierarchy, at least in the same sense as the k-pebble games do: for instance, for every $k \geq 2$, there are NBA $\mathcal{A}$, $\mathcal{B}$ such that $\mathcal{A} \sqsubseteq^k_{\mathsf{stat}} \mathcal{B}$ but $\mathcal{A} \not\sqsubseteq^{k+1}_{\mathsf{stat}} \mathcal{B}$. Indeed, the following two NBA $\mathcal{A}_k$ (left) and $\mathcal{B}_k$ (right) are such a counter-example.



Instead, the static multi-letter games form a lattice which is isomorphic to the one of naturals numbers under the division ordering.

Theorem 6. *For all $k, k' > 0$:*

$$\sqsubseteq^{\mathsf{gcd}(k,k')}_{\mathsf{stat}} \quad\subseteq\quad \sqsubseteq^k_{\mathsf{stat}} \cap \sqsubseteq^{k'}_{\mathsf{stat}} \quad\subseteq\quad \sqsubseteq^k_{\mathsf{stat}} \cup \sqsubseteq^{k'}_{\mathsf{stat}} \quad\subseteq\quad \sqsubseteq^{\mathsf{lcm}(k,k')}_{\mathsf{stat}} \quad .$$

For the purpose of an algorithm that solves static multi-letter games incrementally in k, this result tells that some sequences of values for k could introduce more incompleteness, like e.g. iterating over the powers of 2. Of course, increasing k by 1 in each step would alleviate this problem.

The lower complexity of the static multi-letter games with respect to the multi-pebble games comes at the price of being more incomplete.

Theorem 7. *The following holds.*

1. *For all $k \geq 1$, there are NBA $\mathcal{A}$, $\mathcal{B}$ such that $\mathcal{A} \sqsubseteq^2_{\mathsf{peb}} \mathcal{B}$ (and thus $\mathcal{A} \sqsubseteq_{\mathsf{incl}} \mathcal{B}$), but $\mathcal{A} \not\sqsubseteq^k_{\mathsf{stat}} \mathcal{B}$.*
2. *$\bigcup_{k\geq 1} \sqsubseteq^k_{\mathsf{stat}} \subsetneq \bigcup_{k \geq 1} \sqsubseteq^k_{\mathsf{peb}}$.*

Proof. (1) Consider NBA $\mathcal{A}'$, $\mathcal{B}'$ obtained by slightly modifying NBA $\mathcal{A}$, $\mathcal{B}$ from Ex. 1, by adding an a-loop in the initial state of $\mathcal{A}$ and $\mathcal{B}$. Whatever k is, Refuter can take the a-loop $k-1$ times and then the outgoing a-transition in $\mathcal{A}'$'s initial state. Verifier has to respond with a move that makes him/her commit to the following b or c. In the next round, Refuter can choose ca^{k-1} or ba^{k-1} and Verifier will be stuck. On the other hand, we have $\mathcal{A}' \sqsubseteq^2_{\mathsf{peb}} \mathcal{B}'$.

(2) Assume $\mathcal{A} \sqsubseteq^k_{\mathsf{stat}} \mathcal{B}$. Then Verifier has a winning strategy with $|\mathcal{B}|$ pebbles, since he can mimic a play of the static k-letters game by just pushing all pebbles

following all available choices, except every k rounds in which he only keeps live a single pebble on the state defined by his winning strategy for the k-letters game. $\square$

Another drawback of the static multi-letter games in the perspective of an incremental algorithm is that determining whether the search is hopeless is harder than for the multi-pebble games: the latter are ensured to become stationary, whereas for all $k \geq 1$, it holds that $|\mathcal{G}^k_{\mathsf{stat}}(\mathcal{A}, \mathcal{B})| < |\mathcal{G}^{k+1}_{\mathsf{stat}}(\mathcal{A}, \mathcal{B})|$.

4 The Dynamic Multi-letter Games

In this section we consider a further refinement of the fair simulation game in which, again, Refuter is forced to reveal more information to Verifier. Moreover, Verifier is given additional power in this game which remedies the lack of a linear hierarchy for static games.

Definition. Let $k \geq 1$. The *dynamic k-letter game* is played between Refuter and Verifier on two NBA $\mathcal{A} = (Q_\mathcal{A}, q^\mathcal{A}_{\mathsf{init}}, \Delta_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, q^\mathcal{B}_{\mathsf{init}}, \Delta_\mathcal{B}, F_\mathcal{B})$, similar to the static k-letter game. However, Verifier now can choose how far both players need to advance in each round.

Dynamic k-letter game $\mathcal{G}^k_{\mathsf{dyn}}(\mathcal{A}, \mathcal{B})$	
start:	$q_0 := q^\mathcal{A}_{\mathsf{init}}$, $q'_0 := q^\mathcal{B}_{\mathsf{init}}$, $j := 0$
round i:	(1) Verifier chooses some h with $1 \leq h \leq k$ (2) Refuter chooses some $w = a_1 \dots a_h \in \Sigma^h$ (3) Refuter chooses $q_{j+1}, \dots, q_{j+h}$ such that $q_j \xrightarrow{a_1} q_{j+1} \xrightarrow{a_2} q_{j+2} \xrightarrow{a_3} \dots \xrightarrow{a_h} q_{j+h}$ (4) Verifier chooses $q'_{j+1}, \dots, q'_{j+h}$ such that $q'_j \xrightarrow{a_1} q'_{j+1} \xrightarrow{a_1} q'_{j+2} \xrightarrow{a_2} \dots \xrightarrow{a_h} q'_{j+h}$ (5) $j := j + h$
winner:	same as in fair simulation game

We write $\mathcal{A} \sqsubseteq^k_{\mathsf{dyn}} \mathcal{B}$ if Verifier has a winning strategy for $\mathcal{G}^k_{\mathsf{dyn}}(\mathcal{A}, \mathcal{B})$.

The game $\mathcal{G}^k_{\mathsf{dyn}}(\mathcal{A}, \mathcal{B})$ can be reduced—similar to the k-letter static game—to a parity game of size $\mathcal{O}(|\mathcal{A}| \cdot |\mathcal{B}| \cdot (|\Sigma|^{k+1} + k))$. In the vertices we keep track of the current state of $\mathcal{A}$, $\mathcal{B}$, the h chosen by Verifier, and the word $w = a_1 \dots a_h$ chosen by Refuter. We use the same priority assignment as in the parity game for the static k-letter game.

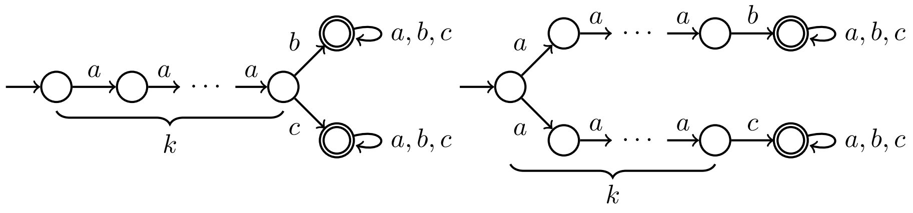
Theorem 8. *For every $k \geq 1$ and NBA $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A} \sqsubseteq^k_{\mathsf{dyn}} \mathcal{B}$ is decidable in time $\mathcal{O}((|\mathcal{A}| \cdot |\mathcal{B}| \cdot (|\Sigma|^{k+1} + k))^3)$.*

Properties. It is quite obvious to see that the 1-letter dynamic game is the same as the 1-letter static game, hence the same as the fair simulation game. Therefore we also have $\sqsubseteq^1_{\mathsf{dyn}} = \sqsubseteq_{\mathsf{fair}}$. As with the static games, the dynamic games approximate language inclusion. However, the dynamic games do form a hierarchy similar to the k-pebble games.

Theorem 9. *The following holds.*

$$\sqsubseteq^1_{\mathsf{dyn}} \subsetneq \sqsubseteq^2_{\mathsf{dyn}} \subsetneq \sqsubseteq^3_{\mathsf{dyn}} \subsetneq \cdots \subsetneq \bigcup_{k\geq 1} \sqsubseteq^k_{\mathsf{dyn}} \subsetneq \sqsubseteq_{\mathsf{incl}}$$

Proof. By definition it is clear that $\sqsubseteq^k_{\mathsf{dyn}} \subseteq \sqsubseteq^{k+1}_{\mathsf{dyn}}$ for every $k \geq 1$. However for every k, there are NBA $\mathcal{A}$, $\mathcal{B}$ with $\mathcal{A} \sqsubseteq^{k+1}_{\mathsf{dyn}} \mathcal{B}$ but $\mathcal{A} \not\sqsubseteq^k_{\mathsf{dyn}} \mathcal{B}$ as we consider the following two NBA $\mathcal{A}$ (left) and $\mathcal{B}$ (right).



□

As mentioned above, for $k = 1$, the static and dynamic games coincide. For larger parameters, the dynamic games are nonetheless more powerful than the static ones.

Theorem 10. *For $k \geq 2$, we have $\sqsubseteq^k_{\mathsf{stat}} \subsetneq \sqsubseteq^k_{\mathsf{dyn}}$.*

Proof. By definition, $\sqsubseteq^k_{\mathsf{stat}} \subseteq \sqsubseteq^k_{\mathsf{dyn}}$. To show $\sqsubseteq^k_{\mathsf{dyn}} \not\subseteq \sqsubseteq^k_{\mathsf{stat}}$ for $k \geq 3$, consider any NBA $\mathcal{A}_k$ and $\mathcal{B}_k$ for which $\mathcal{A}_{k-1} \not\sqsubseteq^k_{\mathsf{stat}} \mathcal{B}_{k-1}$ but $\mathcal{A}_{k-1} \sqsubseteq^{k-1}_{\mathsf{stat}} \mathcal{B}_{k-1}$ holds. By the inclusions $\sqsubseteq^{k-1}_{\mathsf{stat}} \subseteq \sqsubseteq^{k-1}_{\mathsf{dyn}} \subseteq \sqsubseteq^k_{\mathsf{dyn}}$, it holds that $\mathcal{A}_{k-1} \sqsubseteq^k_{\mathsf{dyn}} \mathcal{B}_{k-1}$, which ends the proof. The case $k = 2$ is similar. □

Despite being better approximations of language inclusion than static games, dynamic games suffer from the same drawbacks compared to multi-pebble games.

Theorem 11. *The following holds.*

1. *For all $k \geq 1$, there are NBA $\mathcal{A}$, $\mathcal{B}$ such that $\mathcal{A} \sqsubseteq^2_{\mathsf{peb}} \mathcal{B}$ (and thus $\mathcal{A} \sqsubseteq_{\mathsf{incl}} \mathcal{B}$), but $\mathcal{A} \not\sqsubseteq^k_{\mathsf{dyn}} \mathcal{B}$.*
2. *$\bigcup_{k\geq 1} \sqsubseteq^k_{\mathsf{dyn}} \subsetneq \bigcup_{k\geq 1} \sqsubseteq^k_{\mathsf{peb}}$.*

Although the sequence of games is not stationary, there is an upper bound on the indices k that are worth being tried while looking for a proof of language inclusion.

Theorem 12. *For all NBA $\mathcal{A}, \mathcal{B}$, if there is $k \geq 0$ such that $\mathcal{A} \sqsubseteq^{k}_{\mathsf{dyn}} \mathcal{B}$, then $\mathcal{A} \sqsubseteq^{k_0}_{\mathsf{dyn}} \mathcal{B}$ for $k_0 := 2^{(|\mathcal{A}|+|\mathcal{B}|)^3}$.*

This result could be seen as the indication of a faster convergence of the multi-pebble games towards a fixpoint (since the corresponding upper bound for pebble games is "only" $|B|$). However, these upper bounds are often not tractable in practice, and should be considered with some care.

To conclude, it may be noticed that typical examples are such that $|\Sigma| \ll |\mathcal{B}|$. In this context, the size of dynamic games, like the ones for static games, is expected to grow much slower than those of multi-pebble games. Therefore, the next section compares these games from an experimental point of view.

5 Experiments

We implemented the three incremental inclusion tests using OCaml and the PGSolver library [8] with the recursive algorithm by Zielonka [19]. We evaluated our implementation by comparing it to Rabit [2], a recent Java implementation of optimized Ramsey-based methods (we used the options recommended by the authors, namely `-q -b -rd -fplus -SFS -qr -c -l`). We ran experiments on a machine with 16 Intel Xeon cores at 1.87GH, including experiments with Rabit (note however that Java code naturally tends to run slower than OCaml code). Due to the possible divergence of our algorithms, we fixed a timeout of 1 hour for every incremental inclusion test and every pair of automata. The results as well as the OCaml code are available online.[1]

For our experiments, we used the same benchmarks over which Rabit was tested[2], restricting to pairs of automata $\mathcal{A}, \mathcal{B}$ for which language inclusion can be expected (namely Rabit does not report that $\mathcal{A} \not\sqsubseteq_{\mathsf{incl}} \mathcal{B}$). Rabit benchmarks were obtained from (1) several mutual exclusion protocols with possible errors injected into the code, and (2) the Tabakov-Vardi random model [16], parametrized by the number of states of the automata, the transition density (d_{tr}), and the acceptance density (d_{acc}).

The results for the benchmarks on protocol verification are summarised in Figure 1. The performances of all algorithms are unsurprisingly very similar for the cases where fair simulation holds (k=1). For the cases where fair simulation does not hold, Rabit tends to be the better tool. An exception is BakeryV2, for which Rabit times out, whereas multi-letter simulations perform quite well.

The Tabakov-Vardi benchmarks consist of (1) 4000 pairs of automata with 15 states, d_{tr} ranging from 1.5 to 3, and d_{acc} from 0.1 to 1.0, (2) 100 pairs of automata with 30 states each, $d_{\mathsf{tr}} = 2$, $d_{\mathsf{acc}} = 0.1$, and (3) 100 pairs of automata with 50 states each, $d_{\mathsf{tr}} = 3$, $d_{\mathsf{acc}} = 0.6$. For these parameters one can expect a substantial amount of positive instances for the language inclusion problem, so-called "hard" inclusion in case (2) and easy but non-trivial inclusion in case (3) [2]. We call a method successful on a pair of automata if it can show language

[1] see `http://carrick.fmv.informatik.uni-kassel.de/~milka/iit`

[2] see `http://languageinclusion.org/CONCUR2011`

mutual exclusion protocols

	multi-letter				multi-pebble		Rabit
	k	dynamic	k	static	k	pebble	
Mcs	1	22.94s	1	23.48s	1	25.41s	39.00s
FischerV2	1	0.07s	1	0.07s	1	0.07s	0.09s
Peterson	1	0.01s	1	0.01s	1	0.01s	0.03s
Bakery	1	7.22s	1	7.16s	1	7.23s	4.43s
Phils	1	0.02s	1	0.02s	1	0.02s	0.11s
Fischer	1	40.80s	1	47.30s	1	47.37s	3.41s
FischerV3	-	>1h	-	>1h	-	>1h	7.63s
FischerV4	-	>1h	-	>1h	-	>1h	2136.70s
BakeryV2	2	36.01s	2	7.06s	-	>1h	>1h

random NBA

	size = 30, $d_{\mathsf{acc}} = 0.1$, $d_{\mathsf{tr}} = 2$				size = 50, $d_{\mathsf{acc}} = 0.6$, $d_{\mathsf{tr}} = 3$			
	dynamic	static	pebble	Rabit	dynamic	static	pebble	Rabit
time	59.12s	52.78s	18.22s	1655.84s	0.55s	0.52s	2.09s	127.53s
success %	80%	82%	86%	58%	100%	100%	100%	100%
average k	3.66	4.33	1.93	-	1.01	1.01	1.01	-

Fig. 1. Experimental comparisons between multi-letter simulations and other methods

inclusion before the timeout, and for fixed size, d_{acc}, and d_{tr}, its percentage of success is the rate of successful pairs of automata over all pairs for which one may expect language inclusion. The results for sizes 30 and 50 are given in Figure 1, too. Any incremental inclusion test is almost always two orders of magnitudes faster than Rabit. Incremental inclusion tests are also more successful in general, although the percentage of success of multi-letter games may fall to $\sim 50\%$ for automata of size 15 with timeout 10 minutes (see detailed benchmarks). The pebble game typically explores smaller values of k only. It heavily depends on the size of the automata, in contrast to the multi-letter game.

The experiments demonstrate that incremental inclusion testing is a very reasonable heuristic for proving language inclusion. This heuristic, when it succeeds, is still comparable to the recently optimised Ramsey-based approach, and may perform well when simulation does not hold ($k > 1$).

In order to tackle the lack of completenes, it is tempting to try to combine incremental inclusion and the Ramsey-based approach. It remains to be seen whether or not the results of the multi-letter simulation tests can be used for quotienting. We aim to develop a combination of the two approaches in the future.

References

1. Abdulla, P.A., Chen, Y.-F., Clemente, L., Holík, L., Hong, C.-D., Mayr, R., Vojnar, T.: Simulation Subsumption in Ramsey-Based Büchi Automata Universality and Inclusion Testing. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 132–147. Springer, Heidelberg (2010)

2. Abdulla, P.A., Chen, Y.-F., Clemente, L., Holík, L., Hong, C.-D., Mayr, R., Vojnar, T.: Advanced Ramsey-Based Büchi Automata Inclusion Testing. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 187–202. Springer, Heidelberg (2011)
3. Clemente, L., Mayr, R.: Multipebble Simulations for Alternating Automata (Extended Abstract). In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 297–312. Springer, Heidelberg (2010)
4. Dill, D.L., Hu, A.J., Wong-Toi, H.: Checking For Language Inclusion Using Simulation Preorders. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 255–265. Springer, Heidelberg (1992)
5. Doyen, L., Raskin, J.F.: Antichains for the automata-based approach to model-checking. Logical Methods in Computer Science 5(1) (2009)
6. Etessami, K.: A Hierarchy of Polynomial-Time Computable Simulations for Automata. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 131–144. Springer, Heidelberg (2002)
7. 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)
8. Friedmann, O., Lange, M.: Solving Parity Games in Practice. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer, Heidelberg (2009)
9. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
10. Henzinger, T.A., Kupferman, O., Rajamani, S.K.: Fair simulation. Inf. Comput. 173(1), 64–81 (2002)
11. Jurdziński, M.: Small Progress Measures for Solving Parity Games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000), `http://dl.acm.org/citation.cfm?id=646514.695804`
12. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Transactions on Computational Logic 2(3), 408–429 (2001)
13. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. Theor. Comput. Sci. 32, 321–330 (1984)
14. Safra, S.: On the complexity of ω-automata. In: Proc. 29th Symp. on Foundations of Computer Science, FOCS 1988, pp. 319–327. IEEE (1988)
15. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. Journal of Computer and System Sciences 4, 177–192 (1970)
16. Tabakov, D., Vardi, M.Y.: 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. Thomas, W.: Complementation of Büchi automata revisited. In: Karhumäki, J., et al. (eds.) Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa, pp. 109–122. Springer (1999)
18. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proc. 1st Symp. on Logic in Computer Science, LICS 1986, pp. 332–344. IEEE, Washington, DC (1986)
19. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. TCS 200(1-2), 135–183 (1998)

On Bounded Languages and Reversal-Bounded Automata

Oscar H. Ibarra[1,⋆] and Bala Ravikumar[2]

[1] Department of Computer Science
University of California, Santa Barbara, CA 93106, USA
ibarra@cs.ucsb.edu

[2] Department of Computer & Engineering Science
Sonoma State University, Rohnert Park, CA 94928, USA
ravi@cs.sonoma.edu

Abstract. Bounded context-free languages have been investigated for nearly fifty years, yet they continue to generate interest as seen from recent studies. Here, we present a number of results about bounded context-free languages. First we give a new (simpler) proof that every context-free language $L \subseteq w_1^* w_2^* ... w_n^*$ can be accepted by a PDA with at most $2n-3$ reversals. We also introduce new collections of bounded context-free languages and present some of their interesting properties. Some of the properties are counter-intuitive and may point to some deeper facts about bounded CFL's. We present some results about semilinear sets and also present a generalization of the well-known result that over a one-letter alphabet, the family of context-free and regular languages coincide.

Keywords: context-free language (CFL), nondeterministic pushdown automaton (NPDA), reversal-bounded, semilinear set, stratified linear set.

1 Introduction

The class of context-free languages (CFL) is one of the most important families of languages because of the nice mathematical properties they exhibit and because of their wide-ranging applications. Bounded CFL's [2] are interesting since they admit faster parsing algorithms and many problems that are undecidable for general CFL's are decidable for the bounded CFL's. Also many well-known examples and counter-examples for CFL's are bounded, for example, the standard example of an inherently ambiguous language is a bounded CFL [4]. Some recent work on bounded CFL's include [7], [6] etc. Here we present some properties of bounded context-free languages.

We first show that every context-free language $L \subseteq w_1^* w_2^* ... w_n^*$ can be accepted by a PDA with at most $2n-3$ reversals. This result was also recently shown by [7], but our proof is simpler and is based on a PDA while their proof is based

⋆ Supported in part by NSF Grants CCF-1143892 and CCF-1117708.

on context-free grammars. Then, we introduce a number of bounded languages and present many of their properties. Some of these observations are unexpected. For example, listed below are three languages:

$B_1 = \{a_1^{r_1}a_2^{r_2}a_3^{r_3} \mid r_1 + r_2 \geq r_3,\ r_2 + r_3 \geq r_1,\ r_3 + r_1 \geq r_2\}$.
$B_2 = \{a_1^{n_1+n_2}a_2^{n_2+n_3}a_3^{n_3+n_1} \mid n_1, n_2, n_3 \geq 0\}$.
$B_3 = \{a_1^{r_1}a_2^{r_2}a_3^{r_3}a_4^{r_4} \mid r_1 + r_3 = r_2 + r_4\}$.

Which of the above languages and/or their complements are context-free? What is the (minimum) number of reversals that are needed to accept them? The reader can check their intuition by looking at Section 4 where several such languages are introduced and their properties discussed.

In Section 5, we briefly study the class of semilinear languages and provide a characterization for it in terms of reversal-bounded counter machines. In Section 6, we present a generalization of the well-known result that over a one-letter alphabet, the family of context-free and regular languages coincide. We also present some results about multitape NDPA's with reversal bounded counters. We conclude with some open problems in Section 7.

Due to page limit, most of the proofs are omitted here. A full version can be requested from the authors.

2 Preliminaries

Let N be the set of natural numbers and $n \geq 1$. $Q \subseteq N^n$ is a *linear set* if there is a vector c in N^n (the constant vector) and a set of periodic vectors $V = \{v_1, \ldots, v_r\}$, $r \geq 0$, each v_i in N^n such that $Q = \{c + t_1v_1 + \cdots + t_rv_r \mid t_1, \ldots, t_r \in N\}$. We denote this set as $Q(c, V)$. A finite union of linear sets is called a *semilinear set.*

A linear set $Q(c, V) \subseteq N^n$ is said to be *stratified* if:

1. Every $v \in V$ has at most two nonzero components, and
2. There exist no integers i, j, k, l with $1 \leq i < j < k < l \leq n$ and no vectors $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$ in V such that $u_iv_ju_kv_l \neq 0$.

A finite union of stratified linear sets is called a *stratified semilinear set.*

Let $\Sigma = \{a_1, \ldots, a_n\}$. For $w \in \Sigma^*$, let $|w|$ be the number of letters (symbols) in w, and $|w|_{a_i}$ denote the number of occurrences of a_i in w. The *Parikh map* $P(w)$ of w is the vector $(|w|_{a_1}, \ldots, |w|_{a_n})$; similarly, the Parikh image of a language L is defined as $P(L) = \{P(w) \mid w \in L\}$.

It is known that the Parikh map of a language L accepted by an NPDA (i.e., L is context-free) is an effectively computable semilinear set [8]. A useful generalization of this result for showing the decidability of a wide-range of problems is to extend this result to a larger class of machines. One such class is the class of NPDA's augmented with reversal-bounded counters. A machine M in this class has a pushdown stack, together with a finite set of counters. Each counter can store a single symbol a other than the bottom of the stack symbol. At each move, based on the symbol read by the input head, and the symbol on the top of stack, and counter status (being 0 or non-zero) of each counter, the machine can choose one of a finite number of choices and execute it. This involves applying a move on the input tape (either stay or move right one position), change the state, remove the top of the stack symbol and push a string on the stack, and update each counter (by adding 0, +1 or -1 a's). The fact that the counter is reversal-bounded means that the number of times the counter can change from increasing mode (during which the counter value never decreases) to decreasing mode (during which the counter value never increases) is bounded by a constant, independent of the input length. Note that we place no such restriction on the pushdown stack.

The following result was shown in [5]:

Theorem 1.

1. *If $L \subseteq \Sigma^*$ is accepted by an NPDA with reversal-bounded counters, then $P(L)$ is an effectively computable semilinear set.*
2. *If $L \subseteq w_1^* \cdots w_n^*$ is accepted by an NPDA with reversal-bounded counters (where $w_1, \ldots, w_n$ are nonnull strings), then $Q_L = \{(i_1, \ldots, i_n) \mid w_1^{i_1} \cdots w_n^{i_n} \in L\}$ is an effectively computable semilinear set.*

A language L is *letter-bounded* if it is a subset of $a_1^* \cdots a_n^*$ for some distinct letters (symbols) $a_1, \ldots, a_n$. L is *bounded* if it is a subset of $w_1^* \cdots w_n^*$ for some (not necessarily distinct) nonnull strings $w_1, \ldots, w_n$. The following characterizations of letter-bounded context-free languages (CFLs) are from [2].

Theorem 2.

1. *Let $\Sigma = \{a_1, \ldots, a_n\}$, $n \geq 3$. Each CFL $L \subseteq a_1^* \cdots a_n^*$ is a finite union of sets of the following form:*

$$M(D, E, F) = \{a_1^i x y a_n^j \mid a_1^i a_n^j \in D, x \in E, y \in F\},$$

   where $D \subseteq a_1^* a_n^*$, $E \subseteq a_1^* \cdots a_q^*$, $F \subseteq a_q^* \cdots a_n^*$, $1 < q < n$, are CFLs. Conversely, each finite union of sets of the form $M(D, E, F)$ is a CFL $L \subseteq a_1^* \cdots a_n^*$.
2. *A language $L \subseteq a_1^* \cdots a_n^*$, $n \geq 2$, is a CFL if and only if its Parikh map $P(L) \subseteq N^n$ is a stratified semilinear set (i.e., a finite union of stratified linear sets).*

3 Characterization of Bounded CFLs by Reversal-Bounded NPDAs

We give two constructions to show that every CFL $L \subseteq a_1^* \cdots a_n^*$ can be accepted by a reversal-bounded NPDA. The first construction is simple, but yields an upper bound of $2^n - 1$ on stack reversals. The second construction gives an upper bound of $2n - 3$ on the reversals and there are examples for which the construction achieves this bound.

We begin with the following lemma which is easily verified (see Corollary 7 for a stronger result).

Lemma 3. *Every context-free language $L \subseteq a_1^* a_2^*$ can be accepted by a 1 reversal NPDA.*

Theorem 4. *Every CFL $L \subseteq a_1^* ... a_n^*$ can be accepted by a $2^{n-1} - 1$ reversal-bounded NPDA.*

We now give a construction that improves the upper bound on the stack reversals.

Theorem 5. *Let $L \subseteq a_1^* \cdots a_n^*$ be a CFL. Then we can construct a $2n - 3$ reversal-bounded NPDA accepting L. Moreover, there are examples for which the construction achieves the bound $2n - 3$ for every $n \geq 2$.*

Proof. From Theorem 2, part 2, $L \subseteq a_1^* \cdots a_n^*$ is a CFL if and only if its Parikh map $P(L)$ is a stratified semilinear set (i.e., finite union of stratified linear sets.) Since r-reversal bounded NPDA languages (for any $r \geq 0$) are closed under union, it is sufficient to show that a language L whose Parikh map is a stratified linear set is accepted by a $2n - 3$ reversal-bounded NPDA.

Let Q be a linear set generated by the constant vector $c = (c_1, \ldots, c_n)$ and periodic vectors in the set V.

For $1 \leq i \leq n$, let $V_i = \{v \mid v \in V$, and v has only one non-zero component: the i^{th} component $\}$.

For $1 \leq i, j \leq n$, $i \neq j$, let $V_{ij} = \{v \mid v \in V$, and v has two non-zero components: the i^{th} and j^{th} components $\}$.

We construct an NPDA M that accepts $L = \{a_1^{i_1} \cdots a_n^{i_n} \mid (i_1, \ldots, i_n) \in Q\}$. In addition to the bottom of the stack symbol, M has a pushdown symbol T_{ij} for $1 \leq i < j \leq n$.

Given $a_1^{i_1} \cdots a_n^{i_n}$, M processes each a_i-segment, for $i = 1, \ldots, n$, as follows:

1. M reads c_i a_i's on the input, where c_i is the i^{th} component of the constant vector $c = (c_1, \ldots, c_n)$ of Q.
2. If $V_i = \varnothing$, this step is skipped.

   For each $v \in V_i$ if $v = (0, \ldots, 0, d_i, 0, \ldots, 0)$ (where d_i is the i^{th} component), then for nondeterministically chosen $t_v \geq 0$, M does the following t_v times: reads d_i a_i's without changing the stack. (Thus, for each $v \in V_i$, M reads a total of $t_v d_i$ a_i's.)
3. If there are no vectors in V with nonzero components in position i and in position less than i, this step is skipped. In particular, when $i = 1$, this step is skipped.

   Suppose $V_{j_1 i}, \ldots, V_{j_t i}$ are nonempty and $1 \leq j_1 < ... < j_t < i$. Then M does the following for $k = t, t-1, \ldots, 1$ in this order:

   For each $v \in V_{j_k i}$, if $v = (0 \ldots, 0, d_{j_k}, 0, \ldots, 0, d_i, 0, \ldots, 0)$ (i.e., the nonzero components are in the j_k^{th} and i^{th} positions), then M pops the stack such that for each $T_{j_k i}$ it pops, it reads an a_i. (Thus, for each $v \in V_{j_k i}$, M pops a total of $t_v d_{j_k}$ $T_{j_k i}$'s that have been stored earlier and reads a total of $t_v d_{j_k}$ a_i's.)

4. If there are no vectors in V with nonzero components in position i and in position greater than i, this step skipped.

   Suppose $V_{ij_1}, \ldots, V_{ij_t}$ are nonempty and $i < j_1 < ... < j_t$. Then M does the following for $k = t, t-1, \ldots, 1$ in this order:

   For each $v \in V_{ij_k}$, if $v = (0 \ldots, 0, d_i, 0, \ldots, 0, d_{j_k}, 0, \ldots, 0)$ (i.e., the nonzero components are in positions i_{th} and j_k^{th}), then for nondeterministically chosen $t_v \geq 0$, M does the following: reads d_i a_i's and stack d_{j_k} T_{ij_k}'s on the pushdown. (Thus, for each $v \in V_{ij_k}$, M reads a total of $t_v d_i$ a_i's and stacks a total of $t_v d_{j_k}$ T_{ij_k}'s on the pushdown.)

After the four steps above, M has completed processing the a_i-segment and should be reading the first a_{i+1}. (If not, M rejects the input and halts.) M then proceeds to process the a_{i+1}-segment. When all the a_i-segments have been processed successfully, M accepts the input and halts.

We will now sketch a proof that M accepts L. The basic idea is the following: let $w = a_1^{i_1} a_2^{i_2} ... a_n^{i_n}$ be a string in L. Then $(i_1, i_2, ..., i_n) = c + p_1 v_1 + ... + p_m v_m$, where c is the constant vector and $v_1, \ldots, v_m$ are the periodic vectors. This involves checking that $i_j = c_j + \sum_{r=1}^{m} p_r v_{jr}$, where c_r is the r-th component of c and v_{jr} is the r-th component of v_j. Since V is stratified, there are at most two non-zero components in any vector v_j.

M verifies the above equation by guessing the numbers p_1, ..., p_m and keeping the expression $\sum_{r=1}^{m} p_r v_{jr}$ on the stack while reading the first nonzero component of the vectors whose second nonzero component is j, and matching this value with the value i_j by popping one symbol for each input symbol a_j read. (The vectors with only one non-zero component are handled using the finite-control.)

Next, let us determine the number of stack reversals M makes on an input string in $a_1^* \cdots a_n^*$. For each a_i-segment, $2 \leq i \leq n-1$, M may make a sequence of pops followed by a sequence of pushes. Hence, the number of alternations from popping to pushing and vice-versa for these $n-2$ segments is $2n-5$. (Note that if $n = 2$, the number is 0.) Now the a_1-segment and the a_n segments contribute a pushing sequence sequence and a popping sequence. Thus, in total, there can be $2n - 5 + 2 = 2n - 3$ alternations between popping and pushing and vice-versa or, equivalently, M makes $2n-3$ stack reversals. □

To see that the $2n-3$ bound on the stack reversal is achievable, consider the CFL $L^n_{cycle} = \{a_1^{i_1+i_2} a_2^{i_2+i_3} \cdots a_{n-1}^{i_{n-1}+i_n} a_n^{i_1+i_n} \mid i_1, \ldots, i_n \geq 0\}$. Clearly, the Parikh map of L^n_{cycle} is a stratified linear set with constant vector $c = (0, \ldots, 0)$ and set of periodic vectors $V = \{(1, 0, \ldots, 0, 1), (1, 1, 0, \ldots, 0), (0, 1, 1, 0, \ldots, 0),$
$(0, 0, 1, 1, 0, \ldots, 0), \ldots, (0, \ldots, 0, 1, 1, 0), (0, \ldots, 0, 1, 1)\}$. It is easy to verify that the NPDA accepting L^n_{cycle} using the construction described above will make $2n-3$ reversals. In fact, later we will show that this bound cannot be improved in general since there are n-bounded CFL's that cannot be accepted by a PDA with fewer than $(2n-3)$ reversals.

Corollary 6. *Let $L \subseteq w_1^* \cdots w_n^*$ be a CFL, where $w_1, \ldots, w_n$ are nonnull strings. Then L can be accepted by a $2n-3$ reversal-bounded NPDA, and this reversal-bound is tight.*

For the case when $n = 2$, we have:

Corollary 7. *Every CFL $L \subseteq w_1^* w_2^*$ can be accepted by a 1-reversal counter machine. (Here a counter machine refers to a NPDA in which the stack alphabet consists of two symbols one of which is used only as the bottom of the stack symbol.)*

Malcher and Pighizzini [7] show that the $2n - 3$ reversal-bound in Theorem 5 is tight for a different candidate family L_k:

Theorem 8. *For any $k \geq 1$,*

$$L_k = \{a_1^{i_1+i_2} a_2^{i_2+i_3} \cdots a_{n-1}^{i_{n-1}+i_n} a_n^{i_n} \mid i_1 = 2^k, i_2, \ldots, i_n \geq 1\}$$

cannot be accepted by any NPDA in less than $2n - 3$ reversals.

We note that the result in [7] gave a lower bound of $n - 1$ turns which, when using our definition of reversal-bound, corresponds to the lower bound of $2n - 3$. (Basically, they count only a down-turn as a reversal, while we count both up-turns and down-turns as reversals.)

4 L_{cycle}^n and Related Languages

The language $L_{cycle}^n = \{a_1^{i_1+i_2} a_2^{i_2+i_3} ... a_{n-1}^{i_{n-1}+i_n} a_n^{i_n+i_1} \mid i_1, ..., i_n \geq 0\}$ has some interesting characteristics, which we explore in this section. A related language has been studied in [7].

We first introduce some notation. For an odd integer $n \geq 3$, let $C = (p_1, \ldots, p_n)$, and

$C_1 = (p_1, p_2, \ldots, p_n)$
$C_2 = (p_2, p_3, \ldots, p_n, p_1)$
$C_3 = (p_3, p_4, \ldots, p_n, p_1, p_2)$
....
$C_i = (p_i, p_{i+1}, \ldots, p_n, p_1, \ldots, p_{i-1})$

Thus, C_i is the i-th circular shift of C.

Suppose $C_i = (p_{i_1}, p_{i_2}, \ldots, p_{i_n})$, and the p_{i_j}'s are nonnegative integers. Denote by $S(C_i) = p_{i_1} - p_{i_2} + p_{i_3} - \cdots + p_{i_{n-2}} - p_{i_{n-1}} + p_{i_n}$. For example, $S((3, 6, 7)) =$ 3 - 6 + 7 = 4. Consider the following four collections of languages:

1. $L_{cycle}^n = \{a_1^{i_1+i_2} a_2^{i_2+i_3} ... a_{n-1}^{i_{n-1}+i_n} a_n^{i_n+i_1} \mid i_1, ..., i_n \geq 0\}$, for any $n \geq 3$.
2. $L_1^n = \{a_1^{p_1} a_2^{p_2} ... a_n^{p_n} \mid p_1 + ... + p_n$ is even, $p_1 - p_2 + p_3 - ... - p_{n-2} + p_{n-1} - p_n = 0\}$, for any even $n \geq 2$.
3. $L_2^n = \{a_1^{p_1} a_2^{p_2} ... a_n^{p_n} \mid p_1 + ... + p_n$ is even, $S(C_i) \geq 0$ for $1 \leq i \leq n\}$, for any odd $n \geq 3$
4. $L_3^n = \{a_1^{p_1} a_2^{p_2} ... a_n^{p_n} \mid S(C_i) \geq 0$ for $1 \leq i \leq n\}$, for any odd $n \geq 3$ (note that the the condition $p_1 + ... + p_n$ is even is no longer assumed in L_3^n).

We will show the following in this section:

1. For $n = 2$ and 4: $L^n_{cycle} = L^n_1$, and it and its complement can be accepted by a $2n - 3$ reversal-bounded deterministic counter machine.
2. For any even $n \geq 6$, $L^n_{cycle} \neq L^n_1$. (Thus the equivalence $L^n_{cycle} = L^n_1$ holds only for $n = 2$ and 4.)
3. For any odd $n \geq 3$, $L^n_{cycle} = L^n_2$, and it can be accepted by an unambiguous NPDA (but not likely by any counter machine, even if it is allowed to be ambiguous and there is no restriction on the reversals).
4. For any odd $n \geq 3$, the complement of L^n_{cycle} (= complement of L^n_2) can be accepted by a $2n - 3$ reversal-bounded counter machine.
5. For any odd $n \geq 3$, L^n_3 can be accepted by an unambiguous $2n - 3$ reversal bounded NPDA.
6. For any odd $n \geq 3$, the complement of L^n_3 can be accepted by a $2n - 3$ reversal-bounded counter machine.

First we consider even n and determine the connection between L^n_{cycle} and L^n_1.

Lemma 9. *For $n = 2$ and 4, $L^n_{cycle} = L^n_1$.*

For $n = 2$ and 4, L^n_{cycle} ($= L^n_1$) is context-free. The reason is that the construction in the proof of Theorem 5 applies for both odd n and even n. In fact, it and its complement can be accepted by $2n - 3$ reversal-bounded deterministic counter machines, as shown in the next theorem.

Theorem 10. *For any even $n \geq 2$, L^n_1 and $\overline{L^n_1}$ can be accepted by $2n-3$ reversal-bounded deterministic counter machines.*

For $n = 6$ (and larger n), the situation is different as the following shows.

Proposition 11. *For any even $n \geq 6$, $L^n_{cycle} \neq L^n_1$.*

Now, we consider L^n_2 for odd $n \geq 3$.

Lemma 12. *For any odd $n \geq 3$, $L^n_{cycle} = L^n_2$.*

From the above lemma, we have:

Theorem 13. *For any odd $n \geq 3$, $L^n_2 (= L^n_{cycle})$ can be accepted by a $2n - 3$ reversal-bounded unambiguous NPDA.*

Next, we show that the complement of L^n_{cycle} can be accepted by a reversal-bounded counter machine.

Theorem 14. *For odd $n \geq 3$, $\overline{L^n_2}$ ($= \overline{L^n_{cycle}}$) can be accepted by a $2n-3$ reversal-bounded counter machine, M.*

Proof. We construct M to accept the complement of L_2^n. Let w be an input to M. Clearly, M's finite-control can check and accept if w is not in $a_1^* \cdots a_n^*$ or if $w = a_1^{p_1} \cdots a_n^{p_n}$ but $p_1 + \cdots + p_n$ is not even. Now we describe the operation of M when the input has the correct format, but is not in L_2^n. Clearly, w is not in L_2^n if there exists an $1 \leq i \leq n$ such that $S(C_i) < 0$ where $S(C_i)$ is $p_i - p_{i+1} + ... + (-1)^{n-i} p_n + (-1)^{n-i+1} p_1 + ... + (-1)^{n-1} p_{i-1}$. So M simply guesses an i, which gives an expression of the p_j's with arithmetic operations minus and plus, but can be rewritten so that $p_1, p_2, \ldots, p_n$ appear in this order. (For example, for $n = 5$, $S(C_3) = p_3 - p_4 + p_5 - p_1 + p_2 = -p_1 + p_2 + p_3 - p_4 + p_5$.) The sign vectors are built into the finite-control of M. So when M guesses an i, it knows the associated vector, e.g., if it guesses $S(C_3)$, the sign vector $(-, +, +, -, +)$ indicates that the expression to evaluate is $-p_1 + p_2 + p_3 - p_4 + p_5$. Then M scans the input and evaluates the expression deterministically. In order to compute the expression, the machine has to remember the status as either P or N. P means Increment (Decrement) on seeing a term with positive (negative) sign. For N, it is the opposite. When the counter becomes empty in the middle of a block, switch from P to N (and N to P). We note that $S(C_1)$ is the only guess that will require $2n - 3$ reversals. All the others will require fewer reversals. □

Finally, we consider the language L_3^n:

Theorem 15. *for any odd $n \geq 3$:*

1. *$\underline{L_3^n}$ can be accepted by a $2n - 3$ reversal-bounded unambiguous NPDA.*
2. *$\overline{L_3^n}$ can be accepted by a $2n - 3$ reversal-bounded counter machine. Further, for $n = 3$, the counter machine can be made unambiguous.*

A natural question regarding the languages described in this section (namely, L_{cycle}^n, L_1^n, L_2^n etc.) is if the upper-bound of $2n - 3$ on number of reversals presented in our construction above is tight. Recall Theorem 5 in which we presented the candidate language for which Malcher and Pighizzini [7] presented a technique for showing a lower-bound on the number of reversals. They established a tight-bound (matching the upper and lower-bounds) for L_k language that closely resembles L_{cycle}^n. Recall that L_k is:

$L_k = \{a_1^{i_1+i_2} a_2^{i_2+i_3} \cdots a_{n-1}^{i_{n-1}+i_n} a_n^{i_n} \mid i_1 = 2^k, i_2, \ldots, i_n \geq 1\}$. Note that this language is remarkably similar to L_{cycle}^n. The primary difference is that in the former language (L_k), i_1 takes a constant value 2^k while in the latter language (L_{cycle}^n) , i_1 is arbitrary and that the count of the last block is i_n while in the latter, it is the sum $i_n + i_1$. Another (minor) difference is that in the former language, the variables i_2, ..., i_n are required to be greater than 0 while in the latter case, the variables are allowed to take 0 value. In view of the similarity between the two languages, one may guess that the lower-bound of $2n - 3$ carries over to the latter language along with the proof technique.

However, the situation seems to be more subtle in that the proof technique of Malcher and Pighizzini does not extend to L_{cycle}^n. In fact, the bound $2n - 3$ does not even hold for L_{cycle}^n. We will show this at least in the case of even $n > 2$. The smallest such case is $n = 4$ for which the upper-bound presented in Theorem 5 is 5. However, this is not tight. In the following, we will show that L^4_{cycle} can be accepted with three reversals.

Claim: Let $L^4_{cycle}= \{a_1^{n_1+n_2} a_2^{n_2+n_3} a_3^{n_3+n_4} a_4^{n_4+n_1} | n_1, n_2, n_3, n_4 \geq 0\}$. Then L^4_{cycle} can be accepted by a PDA with 3 reversals.

We next generalize the above claim and show the following:

Theorem 16. *For any even $n \geq 2$, $L^n_1 = \{a_1^{p_1} a_2^{p_2} ... a_n^{p_n} \mid p_1 + ... + p_n$ is even, $p_1 - p_2 + p_3 - ... - p_{n-2} + p_{n-1} - p_n = 0\}$ can be accepted by a 3-reversal PDA.*

Next we will show that three reversals are necessary for accepting L^n_1 for all $n > 2$.

Since it can be assumed that a PDA always starts with a PUSH phase and ends with a POP phase, the number of reversals is always an odd number and hence, if we can show that L^n_1 is not linear then it follows that L^n_1 requires three reversals. We need a lemma from [3].

Lemma 17. *Let $L \subseteq a^+b^+c^+c^+$ be such that*
(1) for all $n, r \geq 1$, $a^n b^n c^r d^r \in L$.
(2) if $a^n b^n c^r d^s \in L$, then $r \leq s$ and
(3) if there exist integers $t_1, t_2 \geq 1$ such that $a^n b^m c^r d^s \in L$ for some $m < n$, then $(n - m)t_1 \leq (r + s)t_2$.
Then, L is not a linear context-free language.

Theorem 18. *For every $n > 2$, a PDA accepting L^n_1 requires three reversals.*

Proof. As remarked above, it suffices to show that L^n_1 is not a linear CFL. We will first show this for $n = 4$. We define M^4_1, a language closely related to L^4_1 as follows: $M^4_1 = \{a_1^n a_2^m a_3^r a_4^s \mid n, m, r, s \geq 1,\ n + r = m + s\}$. It is easy to see that L^4_1 is a linear CFL if and only if M^4_1 is. It is easy to check that M^4_1 satisfies the conditions of lemma 17. (The first two conditions are obvious. We can choose $t_1 = t_2 = 1$ so that condition (3) is satisfied.) This shows the claim for $n = 4$. To show that L^n_1 is not linear for larger values n, it suffices to see that the PDA for L^4_1 can be written as an intersection of L^n_1 and a regular set. □

Finally, we note that an analog of Ogden's lemma for linear languages [1] can be used to show that L^3_{cycle} is not linear.

5 Semilinear Languages

A language $L \subseteq w_1^* \cdots w_n^*$ is called a *semilinear language* if the set $Q = \{(i_1, \ldots, i_n) |w_1^{i_1} \cdots w_n^{i_n} \in L\}$ is a semilinear set.

Corollary 19. *The class of semilinear languages over $w_1^* ... w_n^*$ is the closure under intersection of languages accepted by $2n - 3$ reversal-bounded NPDAs.*

Thus a language $L \subseteq w_1^* ... w_n^*$ is semi-linear if and only if $L = \cap_{j=1}^m L_j$ for some languages L_1, ..., L_m such that each L_j can be accepted by $2n - 3$ reversal-bounded NPDA.

From Theorem 1, part 2 and the above corollary, we get:

Theorem 20. *The class of languages over $w_1^* \cdots w_n^*$ accepted by NPDAs with reversal-bounded counters is the closure under intersection of languages accepted by $2n-3$ reversal-bounded NPDAs.*

A resetting NPDA is a special case of a two-way NPDA (with input end markers). The machine starts with the input head on the left end marker with stack containing only a distinguished bottom of the stack symbol, which is never modified. The machine then computes like an NPDA but when the input head reaches the right end marker, it either enters an accepting state eventually, or resets the input head to the left end marker in some state (which need not be the initial state) with stack again containing only the bottom of the stack symbol. THe machine can then make another (one-way)sweep on the input like an NPDA.

Corollary 21. *A language $L \subseteq w_1^* \cdots w_n^*$ is accepted by NPDA with reversal-bounded counters if and only if it is accepted by a resetting NPDA, where the machine makes no more than $2n-3$ stack reversals between resets.*

6 Multitape NPDAs with Reversal-Bounded Counters

It is well-known that any unary language accepted by an NPDA (i.e., unary CFL) is regular (i.e., accepted by an NFA). In this section, we generalize this result.

A set of strings is *1-bounded* if it is a subset of w^* for some non-null string w. In the following, we generalize the notion of a language to a collection of n-tuples of strings. More precisely, we define a language L as $L \subseteq w_1^* \times \cdots \times w_n^*$. This definition is common place in applications such as relational data modeling. Such a n-tuple language can be recognized by a n-tape automaton M in which each of the strings w_j is stored on a read-only input tape. The input $(w_1, w_2, ..., w_n) \in L(M)$ if there is a sequence of moves leading to acceptance from a suitably defined starting configuration.

Theorem 22. *Let $L \subseteq w_1^* \times \cdots \times w_n^*$ be a set of n-tuples accepted by an n-tape NPDA with reversal bounded counters, where $w_1, \ldots, w_n$ are nonnull strings. (Thus, each input tape is over a 1-bounded set). Then L can be accepted by a n-tape NFA.*

Note that the above theorem does not hold when the tapes are no longer 1-bounded. For example, $L = \{(a^i b^i, a^j) \mid i, j \geq 1\}$ is accepted by a 2-tape NPDA, but it cannot be accepted by any 2-tape NFA; otherwise, the projection of L on the first coordinate (which is not regular) could be accepted by a 1-tape NFA.

In the following, we consider a more general n-tuple language L in which each component language is a bounded (rather than a unary) language.

Let $B_1 \times \cdots \times B_n$ will denote a set of tuples, where each $B_i = w_{i1}^* \cdots w_{ik_i}^*$ for some nonnull strings $w_{i1}, \ldots, w_{ik_i}$. If $L \subseteq B_1 \times \cdots \times B_n$, define $Q_L = \{(i_{11}, \ldots, i_{1k_1}, \ldots, i_{n1}, \ldots, i_{nk_n}) \mid (w_{11}^{i_{11}}, \ldots, w_{1k_1}^{i_{1k_1}}, \ldots, w_{n1}^{i_{n1}}, \ldots, w_{nk_n}^{i_{nk_n}}) \in L\}$.

Corollary 23. *Let $L \subseteq B_1 \times \cdots \times B_n$ be accepted by an n-tape NPDA with reversal-bounded counters. If Q_L is a stratified semilinear set, then L can be accepted by a reversal-bounded n-tape NPDA (i.e., the stack is reversal-bounded and there are no counters).*

As stated in Theorem 1, part 2, the Parikh map, $P(L)$, of a language L accepted by an NPDA with reversal-bounded counters is a semilinear set. We will show that this generalizes to multitape machines.

Let $L \subseteq \Sigma_1^* \times \cdots \times \Sigma_n^*$. For $1 \leq i \leq n$, let $\Sigma_i = \{a_{i1}, \ldots, a_{ik_i}\}$. Define the Parikh map of L as $P(L) = \{(|w_1|_{a_{11}}, \ldots, |w_1|_{a_{1k_1}}, \ldots, |w_n|_{a_{n1}}, \ldots, |w_n|_{a_{nk_n}}) \mid (w_1, \ldots, w_n) \in L\}$.

Theorem 24. *If $L \subseteq \Sigma_1^* \times \cdots \times \Sigma_n^*$ is accepted by an n-tape NPDA M with reversal-bounded counters, then P(L) is a semilinear set.*

Suppose there are r distinct symbols in $\Sigma_1 \cup \cdots \cup \Sigma_n : a_1, \cdots, a_r$. Define $P(L)$ as $P(L) = \{(i_1, \ldots, i_r) \mid i_j = |w_1 \cdots w_n|_{a_j}, (w_1, \ldots, w_n) \in L, 1 \leq j \leq r\}$. Clearly by a construction similar to the proof above, $P(L)$ is also a semilinear set.

An n-tape 2NPDA is an NPDA with n two-way input tapes with left and right end markers on each tape. An n-tuple $(x_1, \ldots, x_n)$ is accepted if the machine, when started with its n input heads at the left end of their respective input tapes, eventually enters an accepting state with all input heads at the right end of their respective tapes. When there is no stack, we have an n-tape 2NFA. In the deterministic versions we use 'D' instead of 'N'. These models can be augmented with reversal-bounded counters.

An n-tape machine (of a given type) is *finite-turn* if there is a nonnegative integer c such that for any accepted n-tuple, there is an accepting computation in which on any input tape, the head makes at most c turns (from left-to-right or right-to-left). Note that if $c = 0$, the machine has one-way input tapes.

Proposition 25. *There is a language L accepted by a 2-turn 2DPDA whose Parikh map, $P(L)$, is not semilinear.*

Proof. Let $a, b, \#$ be new symbols, and

$$L = \{a^1ba^5 \cdots ba^{2n-1}\#a^{2n-2}b \cdots ba^7ba^3 \mid n \geq 1\}.$$

Clearly, L can be accepted by a 2-turn 2DPDA whose stack makes 3 reversals, but $P(L)$ is not semilinear. □

However, for finite-turn 2NFAs, the following was shown in [5].

Theorem 26. *If $L \subseteq \Sigma^*$ is accepted by a finite-turn 2NFA with reversal-bounded counters, then $P(L)$ is semilinear.*

The above theorem does not hold for multitape machines, as the next proposition shows.

Proposition 27. *There is a language L accepted by a 2-turn 2-tape 2DFA such that $P(L)$ is not semilinear.*

Proof. Let $L = \{(a^1b^3 \cdots a^{n-1}, c^2d^4 \cdots c^{n-2}d^n) \mid n = 2i, i \geq 1\}$, where a, b, c, d are distinct symbols. Clearly, L can be accepted by a 2-turn 2-tape DFA, but $P(L)$ is not semilinear, since the projection of $P(L)$ on the first coordinate is the set of squares (and semilinear sets are closed under projections). $\square$

In contrast to Proposition 25, for bounded languages, we have:

Theorem 28. *Let $L \subseteq a_{11}^* \cdots a_{1k_1}^* \times \cdots \times a_{n1}^* \cdots a_{nk_n}^*$ (where the a_{ij}'s are distinct symbols) be accepted by a finite-turn n-tape NPDA M with reversal-bounded counters. Then $P(L)$ is a semilinear set.*

Theorem 29. *Let $L \subseteq B_1 \times \cdots \times B_n$ be accepted by a finite-turn n-tape NPDA with reversal-bounded counters. Then L can be accepted by:*

1. *A finite-turn n-tape 2DFA with one reversal-bounded counter.*
2. *An n-tape (one-way) DFA with a finite number of reversal-bounded counters.*

7 Conclusion

We conclude with the following open problems:

1. Show that L^n_{cycle} cannot be accepted by a counter machine for odd values of n.
2. Show that L^3_2 and L^3_3 are not deterministic context-free languages.
3. Show that L^n_2 and L^n_3 are inherently ambiguous for $n > 5$.
4. Show that the number of reversals needed to accept L^m_{cycle} is $(2m - 3)$ for all m except $m = 4$. While it seems unusual for the optimum bound to exhibit an exception like $m = 4$, Proposition 11 makes this conjecture plausible.

References

1. Boonyavatana, R., Slutzki, G.: A generalized ogden's lemma for linear context-free languages. Bulletin of the EATCS 42 (1985)
2. Ginsburg: The Mathematical Theory of Context-Free Languages. McGraw-Hill, New York (1966)
3. Greibach, S.: Linearity is polynomially decidable for real-time pushdown store automata. Information and Control 42 (1979)
4. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley Publishing Company, Engelwood (1979)
5. Ibarra, O.H.: Reversal-bounded multicounter machines and their decision problems. J. Assoc. Comput. Mach. 25, 116–133 (1978)
6. Kutrib, M., Malcher, A.: Finite turns and the regular closure of linear context-free languages. Discrete Applied Mathematics 155, 2152–2164 (2007)
7. Malcher, A., Pighizzini, G.: Descriptional Complexity of Bounded Context-Free Languages. In: Harju, T., Karhumäki, J., Lepistö, A. (eds.) DLT 2007. LNCS, vol. 4588, pp. 312–323. Springer, Heidelberg (2007)
8. Parikh, R.J.: On context-free languages. J. Assoc. Comput. Mach. 13, 570–581 (1966)

Rewrite Closure and CF Hedge Automata*

Florent Jacquemard[2] and Michael Rusinowitch[1]

[1] INRIA Nancy–Grand Est & LORIA UMR
615 rue du Jardin Botanique, 54602 Villers-les-Nancy, France
rusi@loria.fr
[2] INRIA Paris–Rocquencourt & Ircam UMR
1 place Igor Stravinsky, 75004 Paris, France
florent.jacquemard@inria.fr

Abstract. We introduce an extension of hedge automata called bidimensional context-free hedge automata. The class of unranked ordered tree languages they recognize is shown to be preserved by rewrite closure with inverse-monadic rules. We also extend the parameterized rewriting rules used for modeling the W3C XQuery Update Facility in previous works, by the possibility to insert a new parent node above a given node. We show that the rewrite closure of hedge automata languages with these extended rewriting systems are context-free hedge languages.

Introduction

Hedge Automata (HA) are extensions of tree automata to manipulate unranked ordered trees. They appeared as a natural tool to support document validation since the number of children of a node is not fixed in XML documents and the structural information (type) of an XML document can be specified by an HA.

A central problem in XML document processing is *static typechecking*. This problem amounts to verifying at compile time that every output XML document which is the result of a specified query or transformation applied to an input document with a valid input type has a valid output type. However for transformation languages such as the one provided by XQuery Update Facility (XQUF), the output type of (iterated) applications of update primitives are not easy to predict. Another important issue for XML data processing is the specification and enforcement of access policies. A large amount of work has been devoted to secure XML querying. But most of the work focuses on read-only rights, and very few have considered update rights for a model based on XQUF operations [7,3,9]. These works have considered the sensitive problem of access control policy inconsistency, that is, *whether a forbidden operation can be simulated through a sequence of allowed operations.* For instance [9] presents a hospital database example where it is forbidden to rename a patient name in a medical file but the same effect can be obtained by deleting this file and inserting a new one. This example illustrates a so-called *local inconsistency* problem and its detection can be reduced to checking the emptiness of a HA language.

* This work has been partly supported by the grant ANR-12-CORD-0009 "Inedit".

In formal verification of infinite state systems several regular model checking approaches represent sets of configurations by regular languages, transitions by rewrite rules and (approximations of) reachable configurations as rewrite closure of regular languages see e.g. [6,2]. Regular model checking [1] is extended from tree to hedge rewriting and hedge automata in [15], which gives a procedure to compute reachability sets *approximations*. Here we compute exact reachability sets when the configuration sets are represented by context-free hedge automata, hence beyond the regular (HA) ones. These results are interesting for automated verification where reachability sets are not always regular.

To summarize, several XML validation or infinite-state verification problems would benefit from procedures to compute rewrite-closure of hedge languages. We also need decidable formalisms beyond regular tree languages to capture rewrite closures.

Contributions. In [9] we have proposed a model for XML update primitives of XQUF as parameterized rewriting rules of the form: "insert an unranked tree from a regular tree language L as the first child of a node labeled by a". For these rules, we give type inference algorithms, considering types defined by several classes of unranked tree automata. In particular we have considered context-free hedge automata (CFHA, e.g. [8]), a more general class than regular hedge automata and obtained by requiring that the sequences of sibling states under a node to be in a context-free language. In this submission we first introduce a non-trivial extension of context-free hedge languages defined by what we call bidimensional context-free hedge automata (Section 2). This class is more expressive as shown by examples. The class is also shown to be preserved by rewrite closure when applying inverse-monadic rules that are more general than the rules that were considered in [8](Section 3).

Then we extend the parameterized rewriting rules used for modeling XQUF in [9] by the possibility to insert a new parent node above a given node. We show in Section 4 how to compute the rewrite closure of HA languages with these extended rewriting systems. Although the obtained results are more general than [9] the proofs are somewhat simpler thanks to a new uniform representation of vertical and horizontal steps of CFHA. A full version is available at [10].

Related work. [14] presents a static analysis of XML document adaptations, expressed as sequences of XQUF primitives. The authors also use an automatic inference method for deriving the type, expressed as a HA, of a sequence of document updates. The type is computed starting from the original schema and from the XQuery Updates formulated as rewriting rules as in [9]. However differently from our case the updates are applied in parallel in one shot.

1 Preliminaries

We consider a finite alphabet Σ and an infinite set of variables $\mathcal{X}$. The symbols of Σ are generally denoted $a, b, c \ldots$ and the variables x, $y \ldots$ The sets of *hedges* and *trees* over Σ and $\mathcal{X}$, respectively denoted $\mathcal{H}(\Sigma, \mathcal{X})$ and $\mathcal{T}(\Sigma, \mathcal{X})$, are defined recursively as the smallest sets such that: every $x \in \mathcal{X}$ is a tree, if $t_1, \ldots, t_n$ is a finite sequence of trees (possibly empty), then $t_1 \ldots t_n$ is a hedge and if h is a hedge and $a \in \Sigma$, then $a(h)$ is a tree. The empty hedge (case $n \geq 0$ above) is denoted ε and the tree $a(\varepsilon)$ will be simply denoted by a. We use the operator . to denote the concatenation of hedges. A root (resp. leaf) of a hedge $h = (t_1 \ldots t_n)$ is a root node (resp. leaf node, i.e. node without child) of one of the trees $t_1, ..., t_n$. The root node of $a(h)$ is called the *parent* of every root of h and every root of h is called a *child* of the root of $a(h)$.

We will sometimes consider a tree as a hedge of length one, *i.e.* consider that $\mathcal{T}(\Sigma, \mathcal{X}) \subset \mathcal{H}(\Sigma, \mathcal{X})$. The sets of ground trees (trees without variables) and ground hedges are respectively denoted $\mathcal{T}(\Sigma)$ and $\mathcal{H}(\Sigma)$. The set of variables occurring in a hedge $h \in \mathcal{H}(\Sigma, \mathcal{X})$ is denoted $var(h)$. A hedge $h \in \mathcal{H}(\Sigma, \mathcal{X})$ is called *linear* if every variable of $var(h)$ occurs once in h. A *substitution* σ is a mapping of finite domain from $\mathcal{X}$ into $\mathcal{H}(\Sigma, \mathcal{X})$, whose application (written with postfix notation) is extended homomorphically to $\mathcal{H}(\Sigma, \mathcal{X})$. The set $\mathcal{C}(\Sigma)$ of *contexts* over Σ contains the linear hedges of $\mathcal{H}\big(\Sigma, \{x\}\big)$. The application of a context $C \in \mathcal{C}(\Sigma)$ to a hedge $h \in \mathcal{H}(\Sigma, \mathcal{X})$ is defined by $C[h] := C\{x \mapsto h\}$. A *hedge rewriting system* (HRS) $\mathcal{R}$ over a finite unranked alphabet Σ is a set of *rewrite rules* of the form $\ell \to r$ where $\ell \in \mathcal{H}(\Sigma, \mathcal{X}) \setminus \mathcal{X}$ and $r \in \mathcal{H}(\Sigma, \mathcal{X})$; ℓ and r are respectively called left- and right-hand-side (*lhs* and *rhs*) of the rule. Note that we do not assume the cardinality of $\mathcal{R}$ to be finite. A HRS is called ground, resp. linear, if all its *lhs* and *rhs* of rules are ground, resp. linear.

The rewrite relation $\xrightarrow[\mathcal{R}]{}$ of a HRS $\mathcal{R}$ is the smallest binary relation on $\mathcal{H}(\Sigma, \mathcal{X})$ containing $\mathcal{R}$ and closed by application of substitutions and contexts. In other words, $h \xrightarrow[\mathcal{R}]{} h'$, iff there exists a context C, a rule $\ell \to r$ in $\mathcal{R}$ and a substitution σ such that $h = C[\ell\sigma]$ and $h' = C[r\sigma]$. The reflexive and transitive closure of $\xrightarrow[\mathcal{R}]{}$ is denoted $\xrightarrow[\mathcal{R}]{*}$. Given $L \subseteq \mathcal{H}(\Sigma, \mathcal{X})$ and a HRS $\mathcal{R}$, we define the *rewrite closure* of L under $\mathcal{R}$ as $post^*_{\mathcal{R}}(L) := \{h' \in \mathcal{H}(\Sigma, \mathcal{X}) \mid \exists h \in L, h \xrightarrow[\mathcal{R}]{*} h'\}$.

Example 1. Let us consider the following rewrite rules

$$\mathcal{R} = \{p_0(x) \to a.p_1(x), p_1(x) \to p_2(x).c, p_2(x) \to p_0(b(x)), p_2(x) \to b(x)\}.$$

Starting from $p_0 = p_0(\varepsilon)$, we have the following rewrite sequence $p_0 \to a.p_1 \to a.p_2.c \to a.p_0(b).c \to a.a.p_1(b).c \to a.a.p_2(b).c.c \to a.a.p_0(b(b)).c.c \to \ldots$ The trees of the rewrite closure of p_0 under $\mathcal{R}$ which do not contain the symbols p_0, p_1, p_2 is the set of T-patterns of the form $a \ldots a.b(\ldots b(b)).c \ldots c$ with the same number of a, b and c.

2 Bidimensional Context-Free Hedge Automata

A *bidimensional context-free hedge automaton* ($\mathsf{CF^2HA}$) is a tuple $\mathcal{A} = \langle \Sigma, Q, Q^{\mathsf{f}}, \Delta \rangle$ where Σ is a finite unranked alphabet, Q is a finite set of states disjoint from Σ, $Q^{\mathsf{f}} \subseteq Q$ is a set of final states, and Δ is a set of rewrite rules of one of the following form, where $p_1, \ldots, p_n \in Q \cup \Sigma$, $q \in Q$ and $n \geq 0$

$$\begin{array}{ll} p_1(x_1) \ldots p_n(x_n) \to q(x_1 \ldots x_n) & \text{called } \textit{horizontal} \text{ transitions,} \\ p_1\big(p_2(x)\big) \to q(x) & \text{called } \textit{vertical} \text{ transitions.} \end{array}$$

The move relation $\xrightarrow[\mathcal{A}]{}$ between ground hedges of $\mathcal{H}(\Sigma \cup Q)$ is defined as the rewrite relation defined by Δ. The language of a $\mathsf{CF}^2\mathsf{HA}$ $\mathcal{A}$ in one of its states q, denoted by $L(\mathcal{A}, q)$, is the set of ground hedges $h \in \mathcal{H}(\Sigma)$ such that $h \xrightarrow[\mathcal{A}]{*} q$ (we recall that q stands for $q(\varepsilon)$). A hedge is accepted by $\mathcal{A}$ if there exists $q \in Q^{\mathsf{f}}$ such that $h \in L(\mathcal{A}, q)$. The language of $\mathcal{A}$, denoted by $L(\mathcal{A})$ is the set of hedges accepted by $\mathcal{A}$. We shall also consider below the following kind of transitions, which have the same expressiveness as $\mathsf{CF}^2\mathsf{HA}$.

$$\begin{array}{rll} p_1(\delta_1)\ldots p_n(\delta_n) \to q(\delta_1 \ldots \delta_n) & & n > 0 \\ p_1(p_2(\delta_1)) \to q(\delta_1) & & \text{every } \delta_i \text{ is either a variable } x_i \text{ or } \varepsilon \end{array}$$

Example 2. The language of T-patterns over $\Sigma = \{a, b, c\}$, see Example 1, is recognized by $\langle \Sigma, \{q_0, q_1, q_2\}, \{p_0\}, \Delta \rangle$ with $\Delta = \{b(x_1) \to q_0(x_1),\ a.q_0(x_2) \to q_1(x_2),\ q_1(x_1).c \to q_2(x_1),\ q_2(b(x)) \to q_0(x)\}$.

2.1 Related Models

The $\mathsf{CF}^2\mathsf{HA}$ capture the expressiveness of two models of automata on unranked trees: the hedge automaton [11] and the lesser known extension of [12] that we call CFHA. A *hedge automaton* (HA) resp. *context-free hedge automaton* (CFHA) is a tuple $\mathcal{A} = \langle \Sigma, Q, Q^{\mathsf{f}}, \Delta \rangle$ where Σ, Q and Q^{f} are as above, and the transitions of Δ have the form $a(L) \to q$ where $a \in \Sigma$, $q \in Q$ and $L \subseteq Q^*$ is a regular word language (resp. a context-free word language). The language of hedges accepted is defined as for $\mathsf{CF}^2\mathsf{HA}$, using the rewrite relation of Δ.

The CFHA languages form a strict subclass of $\mathsf{CF}^2\mathsf{HA}$ languages. Indeed every CFHA can be presented as a $\mathsf{CF}^2\mathsf{HA}$ with variable-free transitions of the form

$$p_1 \ldots p_n \to q \quad a(q_1) \to q_2 \quad \text{where } a \in \Sigma \text{ and } q_1, q_2 \text{ are states.}$$

It can be shown that the set of T-patterns of Example 2 is not a CFHA language, using a pumping argument on the paths labeled by b.

The HA languages, also called *regular* languages, also form a strict subclass of $\mathsf{CF}^2\mathsf{HA}$ languages. Every HA can indeed be presented as a $\mathsf{CF}^2\mathsf{HA}$ $\mathcal{A} = (\Sigma, Q, Q^{\mathsf{f}}, \Delta)$ with variable-free transitions constrained with a type discipline: $Q = Q_{\mathsf{h}} \uplus Q_{\mathsf{v}}$ and every transition of Δ has one of the forms

$$\varepsilon \to q_{\mathsf{h}} \quad q_{\mathsf{h}}.q_{\mathsf{v}} \to q'_{\mathsf{h}} \quad a(q_{\mathsf{h}}) \to q_{\mathsf{v}} \quad \text{where } q_{\mathsf{h}}, q'_{\mathsf{h}} \in Q_{\mathsf{h}}, q_{\mathsf{v}} \in Q_{\mathsf{v}}, a \in \Sigma.$$

From now on, we shall always consider HA and CFHA presented as $\mathsf{CF}^2\mathsf{HA}$. The following example shows that $\mathsf{CF}^2\mathsf{HA}$ can capture some CF ranked tree languages. Capturing the whole class of CF RTL would require however a further generalization where permutations of variables are possible in the horizontal transitions of $\mathsf{CF}^2\mathsf{HA}$. Such a generalization is out of the scope of this paper.

Example 3. The language $\{h^n(g(a^n(0), b^n(0))) \mid n \geq 1\}$ is generated by the CF ranked tree grammar [4] with non-terminals A and S (S is the axiom) and productions $A(x_1, x_2) \to h\big(A(a(x_1), b(x_2))\big)$, $A(x_1, x_2) \to g(x_1, x_2)$ and $S \to A(0, 0)$. It is also recognized by the $\mathsf{CF}^2\mathsf{HA}$ with transition rules $a(x_1).b(x_2) \to q(x_1.x_2)$, $g(x_1) \to q_0(x_1)$, $q_0(q(x)) \to q_1(x)$, $h(q_1(x)) \to q_0(x)$ (q_0 is final).

2.2 Properties

The class of $\mathsf{CF}^2\mathsf{HA}$ language is closed under union (direct construction by disjoint union of automata) and not closed under intersection or complementation (because CF word languages are defined by $\mathsf{CF}^2\mathsf{HA}$ without vertical transitions).

Property 4. The membership problem is decidable for $\mathsf{CF}^2\mathsf{HA}$.

Proof. Let $h \in \mathcal{H}(\Sigma)$ be a given hedge and $\mathcal{A}$ be a given $\mathsf{CF}^2\mathsf{HA}$. We assume *wlog* that $\mathcal{A}$ is presented as a set Δ of transitions in the above alternative form $p_1(\delta_1)\dots p_n(\delta_n) \to q(\delta_1 \dots \delta_n)$, with $n > 0$, and $p_1(p_2(\delta_1)) \to q(\delta_1)$.

Moreover, we assume that every transition of the form $q_1(x_1) \to q_2(x_1)$, where q_1 and q_2 are states, has been removed, replacing arbitrarily q_1 by q_2 in the *rhs* of the other transitions. Similarly, we remove $q_1 \to q_2$, replacing arbitrarily *rhs*'s of the form q_1 by q_2. All these transformations increase the size of $\mathcal{A}$ polynomially.

Then all the horizontal transitions with $n = 1$ have the form $a(\delta_1) \to q(\delta_1)$, with $a \in \Sigma$. It follows that the application of every rule of Δ strictly reduces the measure on hedges defined as pair (# of occurrences of symbols of Σ, # of occurrences of state symbols), ordered lexicographically. During a reduction of h by Δ, each of the two components of the above measure is bounded by the size of h. It follows that the membership $h \in L(\mathcal{A})$ can be tested in PSPACE. $\square$

Property 5. The emptiness problem is decidable in PTIME for $\mathsf{CF}^2\mathsf{HA}$.

Proof. Let $\mathcal{A} = \langle \Sigma, Q, Q^{\mathsf{f}}, \Delta \rangle$. We use a marking algorithm with two marks: h and v. First, for technical convenience, we mark every symbol in Σ with v. Then we iterate the following operations until no marking is possible (note that the marking is not exclusive: some states may have 2 marks h and v).
For all transition $p_1(x_1)\dots p_n(x_n) \to q(x_1 \dots x_n)$ in Δ such that every p_i is marked, if at least one p_i is marked with v, then mark q with v, otherwise mark q with h.
For all transition $p_1\big(p_2(x)\big) \to q(x)$ in Δ such that p_1 is marked v, if p_2 is marked with v, then mark q with v, otherwise, if p_2 is marked with h, then mark q with h.

The number of iterations is at most $2.|Q|$ and the cost of each iteration is linear in the size of $\mathcal{A}$. Then $q \in Q$ is marked with h only iff there exists $h \in \mathcal{H}(\Sigma)$ such that $h \xrightarrow[\Delta]{*} q$, and it is marked with v iff there exists $C[\,] \in \mathcal{C}(\Sigma)$ such that for all $h \in \mathcal{H}(\Sigma)$, $C[h] \xrightarrow[\Delta]{*} q(h)$. Hence $L(\mathcal{A}) = \emptyset$ iff no state of Q^{f} is marked. $\square$

For comparison, for both classes of HA and CFHA, the membership and emptiness problems are decidable in PTIME, the class of HA languages is closed under Boolean operations and the class of CFHA languages is closed under union but not closed under intersection and complementation, see [11,12,4].

3 Inverse Monadic Hedge Rewriting Systems

A rewrite rule $\ell \to r$ over Σ is called *monadic* (following [13,5]) if $r = a(x)$ with $a \in \Sigma$, $x \in \mathcal{X}$, *inverse-monadic* if $r \to \ell$ is monadic and $r \notin \mathcal{X} \cup \{\varepsilon\}$, and

1-childvar if it contains at most one variable and this variable has no siblings in ℓ and r. Intuitively, every finite, linear, inverse-monadic, 1-childvar HRS can be transformed into a HRS equivalent *wrt* reachability whose rules are inverse of transitions of $\mathsf{CF}^2\mathsf{HA}$. It follows that such HRS preserve $\mathsf{CF}^2\mathsf{HA}$ languages.

Example 6. The HRS of Example 1 is linear, inverse-monadic, and 1-childvar. The closure of the language $\{p_0\}$ is the $\mathsf{CF}^2\mathsf{HA}$ language of T-patterns.

Theorem 7. *Let L be the language of $\mathcal{A}_L \in \mathsf{CF}^2\mathsf{HA}$, and $\mathcal{R}$ be a finite, linear, inverse-monadic, 1-childvar* HRS. *There exists an effectively computable $\mathsf{CF}^2\mathsf{HA}$ recognizing $post^*_{\mathcal{R}}(L)$, of size polynomial in the size of $\mathcal{R}$ and $\mathcal{A}_L$.*

Proof. Let $\mathcal{A}_L = \langle \Sigma, Q_L, Q^{\mathsf{f}}_L, \Delta_L \rangle$, we construct a $\mathsf{CF}^2\mathsf{HA}$ $\mathcal{A} = \langle \Sigma, Q, Q^{\mathsf{f}}, \Delta \rangle$. The state set Q contains all the states of Q_L, one state $\underline{h}$ for every non-variable sub-hedge of a *rhs* of rule of $\mathcal{R}$, one state $\underline{a}$ for each $a \in \Sigma$ and one new state $q \notin Q_L$. For each $p \in Q_L \cup \Sigma$, we note $\underline{p} = \underline{a}$ if $p = a \in \Sigma$ and $\underline{p} = p$ otherwise. Let $Q^{\mathsf{f}} = Q^{\mathsf{f}}_L$ and let Δ_0 contain the following transition rules, where $a \in \Sigma$, $t \in \mathcal{T}(\Sigma, \{x\})$ and $h \in \mathcal{H}(\Sigma, \{x\}) \setminus \{\varepsilon\}$.

$$\begin{array}{rll}
\underline{p_1}(x_1)\ldots\underline{p_n}(x_n) \to q(x_1\ldots x_n) & \text{if } p_1(x_1)\ldots p_n(x_n) \to q(x_1\ldots x_n) \in \Delta_L \\
\underline{p_1}\big(\underline{p_2}(x)\big) \to q(x) & \text{if } p_1\big(p_2(x)\big) \to q(x) \in \Delta_L
\end{array}$$

$$\begin{array}{lll}
\underline{t}(x).\underline{h} \to \underline{t.h}(x) \text{ if } x \in var(t), \underline{t.h} \in Q & a(x) \to \underline{a}(x) & \\
\underline{t}(x).\underline{h} \to q(x) \quad \text{if } x \in var(t), \underline{t.h} \notin Q & a(\underline{h}(x)) \to \underline{a(h)}(x) & \text{if } \underline{a(h)} \in Q \\
\underline{t}.\underline{h}(x) \to \underline{t.h}(x) \text{ if } x \notin var(t), \underline{t.h} \in Q & a(\underline{h}(x)) \to \underline{a}(x) & \text{if } \underline{a(h)} \notin Q \\
\underline{t}.\underline{h}(x) \to q(x) \quad \text{if } x \notin var(t), \underline{t.h} \notin Q & a(q(x)) \to \underline{a}(x) &
\end{array}$$

Finally let $\Delta = \Delta_0 \cup \{\underline{h}(x) \to \underline{a}(x) \mid a(x) \to h \in \mathcal{R}\}$. Let $\ell \in \mathcal{H}(\Sigma)$ be such that $\ell \xrightarrow[\Delta]{*} s(u)$ $(\star)$, with $s \in Q$ and $u \in \mathcal{H}(Q \cup \Sigma)$. We show by induction on the number N of applications of rules of $\Delta \setminus \Delta_0$ in $(\star)$ that there exists $\ell' \in \mathcal{H}(\Sigma)$ such that $\ell' \xrightarrow[\mathcal{R}]{*} \ell$ and moreover, if $s = \underline{h}$, then h matches ℓ', if $s = q$ then ℓ' is not matched by a non-variable subhedge of *rhs* of rule of $\mathcal{R}$ and if $s \in Q_L$, then $\ell' \in L(\mathcal{A}_L, s)$.

If $N = 0$, then the property holds with $\ell' = \ell$ (this can be shown by induction on the length of $(\star)$). If $N > 0$, we can assume that $(\star)$ has the following form.

$$\ell = C[k] \xrightarrow[\Delta_0]{*} C[\underline{h}(v)] \xrightarrow[\Delta\setminus\Delta_0]{} C[\underline{a}(v)] \xrightarrow[\Delta]{} s(u)$$

It follows that h matches k, i.e. there exists w such that $k = h[w]$, and $w \xrightarrow[\Delta_0]{*} v$. Hence $\ell' = C[a(w)] \xrightarrow[\mathcal{R}]{} \ell$, and $\ell' \xrightarrow[\Delta_0]{*} C[a(v)] \xrightarrow[\Delta_0]{} C[\underline{a}(v)] \xrightarrow[\Delta]{} s(u)$. We can then apply the induction hypothesis to ℓ', and immediately conclude for ℓ. □

The following Example 8 illustrates the importance of the 1-childvar and condition in Theorem 7.

Example 8. With the following rewrite rule $a(x) \to c\, a(e\, x\, g)\, d$ we generate from $\{a\}$ the language $\{c^n a(e^n g^n)\, d^n \mid n \geq 1\}$, seemingly not $\mathsf{CF}^2\mathsf{HA}$.

In [8] it is shown that the closure of a HA language under rewriting with a monadic HRS is a HA language. It follows that the backward rewrite closure of a HA language under an inverse-monadic HRS is HA.

4 Update Hedge Rewriting Systems

In this section, we turn to our motivation of studying XQuery Update Facility primitives modeled as parameterized rewriting rules.

Let $\mathcal{A} = \langle \Sigma, Q, Q^{\mathsf{f}}, \Delta \rangle$ be a HA. A hedge rewriting system over Σ parametrized by $\mathcal{A}$ (PHRS) is given by a finite set, denoted $\mathcal{R}/\mathcal{A}$, of rewrite rules $\ell \to r$ where $\ell \in \mathcal{H}(\Sigma, \mathcal{X})$ and $r \in \mathcal{H}(\Sigma \uplus Q, \mathcal{X})$ and symbols of Q can only label leaves of r ($\uplus$ stands for the disjoint union, hence we implicitly assume that Σ and Q are disjoint sets). In this notation, $\mathcal{A}$ may be omitted when it is clear from context or not necessary. The rewrite relation $\xrightarrow[\mathcal{R}/\mathcal{A}]{}$ associated to a PHRS $\mathcal{R}/\mathcal{A}$ is defined as the rewrite relation $\xrightarrow[\mathcal{R}[\mathcal{A}]]{}$ where the HRS $\mathcal{R}[\mathcal{A}]$ is the (possibly infinite) set of all rewrite rules obtained from rules $\ell \to r$ in $\mathcal{R}/\mathcal{A}$ by replacing in r every state $p \in Q$ by a ground hedge of $L(\mathcal{A}, p)$. Note that when there are multiple occurrences of a state p in a rule, each occurrence of p is independently replaced with a hedge in $L(\mathcal{A}, p)$, which can generally be different from one another. Given a set $L \subseteq \mathcal{H}(\Sigma, \mathcal{X})$, we define $post^*_{\mathcal{R}/\mathcal{A}}(L)$ to be $post^*_{\mathcal{R}[\mathcal{A}]}(L)$.
We call *updates* parametrized rewrite rules of the following form

$a(x) \to b(x)$		node renaming	(ren)
$a(x) \to a(u_1\, x\, u_2)$	$u_1, u_2 \in Q^*$	addition of child nodes	(ac)
$a(x) \to v_1\, a(x)\, v_2$	$v_1, v_2 \in Q^*$	addition of sibling nodes	(as)
$a(x) \to b\big(a(x)\big)$		addition of parent node	(ap)
$a(x) \to u$	$u \in Q^*$	node replacement/recursive deletion	(rpl)
$a(x) \to x$		single node deletion	(del)

Note that the particular case of (rpl) of rpl with $u = \varepsilon$ corresponds to the deletion of the whole subtree $a(x)$. In the rest of the paper, a PHRS containing only updates will be called update PHRS (uPHRS).

4.1 Loop-Free uPHRS

In order to simplify the proofs we can reduce to the case where there exists no looping sequence of renaming. This motivates the following definition:

Definition 9. *An* uPHRS $\mathcal{R}/\mathcal{A}$ *is* loopfree *if there exists no sequence* $a_1, \ldots, a_n$ *(*$n > 1$*) such that for all* $1 \leq i < n$*,* $a_i(x) \to a_{i+1}(x) \in \mathcal{R}$ *and* $a_1 = a_n$*.*

Given a uPHRS $\mathcal{R}/\mathcal{A}$, we consider the directed graph G whose set of nodes is Σ and containing an edge $\langle a, b \rangle$ iff $a(x) \to b(x)$ is in $\mathcal{R}$. For every strongly connected component in G we select a representative. We denote by $\hat{a}$ the representative of a in its component and more generally by $\hat{h}$ the hedge obtained from $h \in \mathcal{H}(\Sigma)$ by replacing every function symbol a by its representative $\hat{a}$. We define $\hat{\mathcal{R}}$ to be $\mathcal{R}$ where every rule $\ell \to r$ is replaced by $\hat{\ell} \to \hat{r}$ (if the two members get equal we can remove the rule). We define $\hat{\mathcal{A}}$ analogously.

Lemma 10. *Given an* uPHRS $\mathcal{R}/\mathcal{A}$ *the* uPHRS $\hat{R}/\hat{A}$ *is loopfree and for all* $h, h' \in \mathcal{H}(\Sigma)$ *we have* $h \xrightarrow[\mathcal{R}/\mathcal{A}]{*} h'$ *iff* $\hat{h} \xrightarrow[\hat{R}/\hat{A}]{*} \hat{h}'$*.*

Proof. By induction on the length of derivations. $\square$

4.2 Rewrite Closure

The rest of the section is devoted to the proof of the following theorem of construction of $\mathsf{CF}^2\mathsf{HA}$ for the forward closure by updates.

Theorem 11. *Let $\mathcal{A}$ be a* HA *over Σ, and L be the language of $\mathcal{A}_L \in$* CFHA*, and $\mathcal{R}/\mathcal{A}$ be a loop-free* uPHRS*. There exists an effectively computable* CFHA *recognizing $post^*_{\mathcal{R}/\mathcal{A}}(L)$, of size polynomial in the size of $\mathcal{R}/\mathcal{A}$ and $\mathcal{A}_L$ and exponential in the size of the alphabet Σ.*

The construction of the CFHA works in 2 steps: construction of an initial automaton and completion loop. We shall use the following notion in order to simplify the proof: a CFHA $\langle \Sigma, Q, Q^{\mathsf{f}}, \Delta\rangle$ is called *normalized* if for all $a \in \Sigma$ and $q \in Q$, there exists one unique state of Q denoted q^a such that $a(q^a) \to q \in \Delta$, and moreover, q^a does neither occur in a left hand side of an horizontal transition of Δ nor in a right hand side of a vertical transition of Δ. With some state renaming, every CFHA $\mathcal{A}$ can be transformed in PTIME into a normalized CFHA $\mathcal{A}'$, of size linear in the size of $\mathcal{A}$, and such that $L(\mathcal{A}') = L(\mathcal{A})$.

Initial automaton. Let $\mathcal{A} = \langle \Sigma, Q_{\mathcal{A}}, Q^{\mathsf{f}}_{\mathcal{A}}, \Delta_{\mathcal{A}}\rangle$ and $\mathcal{A}_L = \langle \Sigma, Q_L, Q^{\mathsf{f}}_L, \Delta_L\rangle$. We assume that the state sets $Q_{\mathcal{A}}$ and Q_L are disjoint. First, let us merge $\mathcal{A}$ and $\mathcal{A}_L$ into a CFHA $\mathcal{B} = \langle \Sigma, P, P^{\mathsf{f}}, \Gamma\rangle$ obtained by the normalization of $\langle \Sigma, Q_{\mathcal{A}} \uplus Q_L, Q^{\mathsf{f}}_L, \Delta_{\mathcal{A}} \uplus \Delta_L\rangle$. Below, the states of P will be denoted by the letters p or q. Let P_{in} be the subset of states of P of the form q^a (remember that q^a is a state of P uniquely characterized by $a \in \Sigma$, $q \in P$, since $\mathcal{B}$ is normalized). We assume *wlog* that P_{in} and P^{f} are disjoint and that $\mathcal{B}$ is *clean*, *i.e.* for all $p \in P$, $L(\mathcal{B}, p) \neq \emptyset$.

Next, in a preliminary construction step, we transform the initial automaton $\mathcal{B}$ into a CFHA $\mathcal{A}_0 = \langle \Sigma, Q, Q^{\mathsf{f}}, \Delta_0\rangle$. Let us call *renaming chain* a sequence $a_1, \ldots, a_n$ of symbols of Σ such that $n \geq 1$ for all $1 \leq i < n$, $a_i(x) \to a_{i+1}(x) \in \mathcal{R}$. Since $\mathcal{R}$ is loop-free, the length of every renaming chains is bounded by $|\Sigma|$. The fresh state symbols of Q are defined as extensions of the symbols of $P \setminus P_{\mathsf{in}}$ with renaming chains. We consider two modes for such states: the *push* and *pop* modes, characterized by a chain respectively in superscript or subscript.

$$Q = P \cup \{q_a \mid q^a \in P_{\mathsf{in}}\} \cup \left\{ \begin{array}{c|l} q^{a_1 \cdots a_n} & q \in P \setminus P_{\mathsf{in}}, n \geq 2, \\ q_{a_1 \ldots a_n} & a_1, \ldots, a_n \text{ is a renaming chain} \end{array} \right\}$$

Let $Q^{\mathsf{f}} = P^{\mathsf{f}}$ be the subset of final states. Intuitively, in the state $q^{a_1 \cdots a_n}$, the chain of Σ^+ represents a sequence of renamings, with $\mathcal{R}/\mathcal{A}$, of the parent of the current symbol, starting with a_1 and ending with a_n. Note that the states of P_{in} are particular cases of such states, with a chain of length one. A state $q_{a_1 \ldots a_n}$ will be used below to represent the tree $a_n(q^{a_1 \cdots a_n})$.
The initial set of transitions Δ_0 is defined as follows

$$\begin{aligned} \Delta_0 = \Gamma_h &\cup \{q_{a_1} \to q \mid q_{a_1} \in Q\} \\ &\cup \{a_n\big(q^{a_1 \cdots a_n}\big) \to q_{a_1 \ldots a_n} \mid q^{a_1 \cdots a_n}, q_{a_1 \ldots a_n} \in Q, n \geq 1\} \end{aligned}$$

where Γ_h is the subset of horizontal transitions of Γ. Note that $\mathcal{A}_0$ is not normalized. The following lemma is immediate by construction of Γ and $\mathcal{A}_0$.

Lemma 12. *For all* $q \in Q_{\mathcal{A}}$ *(resp.* $q \in Q_L$*)* $L(\mathcal{A}_0, q) = L(\mathcal{A}, q)$ *(resp.* $L(\mathcal{A}_L, q)$*).*

Proof. Every vertical transition in Γ has the form $a(q^a) \to q$ and can be simulated by the 2 steps $a(q^a) \to q_a \to q$. Moreover, all the states $q^{a_1 \cdots a_n}$ and $q_{a_1 \ldots a_n}$ with $n \geq 2$ are empty for $\mathcal{A}_0$. □

For the construction of $\mathcal{A}'$, we shall complete incrementally Δ_0 into Δ_1, Δ_2,... by adding some transition rules, according to a case analysis of the rules of $\mathcal{R}/\mathcal{A}$. For each construction step $i \geq 0$, we let $\mathcal{A}_i = \langle \Sigma, Q, Q^{\mathsf{f}}, \Delta_i \rangle$.

Automata completion. The construction of the sequence (Δ_i) works by iteration of a case analysis of the rewrite rules of $\mathcal{R}/\mathcal{A}$, presented in Table 1. Assuming that Δ_i is the last set built, we define its extension Δ_{i+1} by application of the first case in Table 1 such that $\Delta_{i+1} \neq \Delta_i$. In the rules of Table 1, $a_1, \ldots, a_n, b$ are symbols of Σ, and u, v are sequences of $Q^*_{\mathcal{A}}$.

Table 1. CFHA Completion

	$\mathcal{R}/\mathcal{A}$ contains	$\Delta_{i+1} = \Delta_i \cup$
(ren)	$a_n(x) \to b(x)$	$\{q^{a_1 \ldots a_n} \to q^{a_1 \ldots a_n b} \mid q^{a_1 \ldots a_n b} \in Q\}$ $\cup \{q_{a_1 \ldots a_n b} \to q_{a_1 \ldots a_n} \mid q_{a_1 \ldots a_n b} \in Q\}$
(ac)	$a_n(x) \to a_n(u\,x\,v)$	$\{u\, q^{a_1 \cdots a_n}\, v \to q^{a_1 \cdots a_n} \mid q^{a_1 \cdots a_n} \in Q\}$
(as)	$a_n(x) \to u\, a_n(x)\, v$	$\{u\, q_{a_1 \ldots a_n}\, v \to q_{a_1 \ldots a_n} \mid q_{a_1 \ldots a_n} \in Q\}$
(ap)	$a_n(x) \to b\big(a_n(x)\big)$	$\{b\big(q_{a_1 \ldots a_n}\big) \to q_{a_1 \ldots a_n} \mid q_{a_1 \ldots a_n} \in Q\}$
(rpl)	$a_n(x) \to u$	$\{u \to q_{a_1 \ldots a_n} \mid q_{a_1 \ldots a_n} \in Q\}$
(del)	$a_n(x) \to x$	$\{q^{a_1 \cdots a_n} \to q_{a_1 \ldots a_n} \mid q^{a_1 \cdots a_n} \in Q\}$

Only a bounded number of rules can be added to the Δ_i's, hence eventually, a fixpoint Δ_k is reached, that we will denote Δ'. We also write $\mathcal{A}'$ for $\mathcal{A}_k$.

The following Lemma 13 shows that the automata computations simulate the rewrite steps, *i.e.* that $L(\mathcal{A}') \subseteq \mathit{post}^*_{\mathcal{R}/\mathcal{A}}(L)$. Let us abbreviate $\mathcal{R}/\mathcal{A}$ by $\mathcal{R}$. We use the notation $h \xrightarrow[\mathcal{R}]{a_1 \ldots a_n} h'$, for a renaming chain $a_1, \ldots, a_n$ ($n \geq 1$), if there exists $h_1, \ldots h_n \in \mathcal{H}(\Sigma)$ such that

$$h = a_1(h_1) \xrightarrow[\mathcal{R}]{*} a_1(h_2) \xrightarrow[\mathsf{ren}]{} a_2(h_2) \xrightarrow[\mathcal{R}]{*} \ldots \xrightarrow[\mathcal{R}]{*} a_{n-1}(h_n) \xrightarrow[\mathsf{ren}]{} a_n(h_n) \xrightarrow[\mathcal{R}]{*} h'$$

where the reductions denoted $\xrightarrow[\mathsf{ren}]{}$ are rewrite steps with rules of $\mathcal{R}/\mathcal{A}$ of type (ren), applied at the positions of a_1,..., a_n, and all the other rewrite steps (denoted $\xrightarrow[\mathcal{R}]{*}$) involve no rule of type (ren).

Lemma 13 (Correctness). *For all* $h \in \mathcal{H}(\Sigma)$*,*

i. if $h \xrightarrow[\mathcal{A}']{*} q_{a_1 \ldots a_n}$*, with* $n \geq 1$*, then there exists* $h_1 \in \mathcal{H}(\Sigma)$ *such that* $a_1(h_1) \xrightarrow[\mathcal{B}]{*} q$ *and* $a_1(h_1) \xrightarrow[\mathcal{R}]{a_1 \ldots a_n} h$*,* *ii. if* $h \xrightarrow[\mathcal{A}']{*} q^{a_1 \cdots a_n}$, *with* $n \geq 1$, *then there exists* $h_1 \in \mathcal{H}(\Sigma)$ *such that* $h_1 \xrightarrow[\mathcal{B}]{*} q^{a_1}$, *and* $a_1(h_1) \xrightarrow[\mathcal{R}]{a_1 \ldots a_n} a_n(h)$,

iii. if $h \xrightarrow[\mathcal{A}']{*} q \in P \setminus P_{\mathsf{in}}$, *then there exists* $h' \in \mathcal{H}(\Sigma)$ *such that* $h' \xrightarrow[\mathcal{B}]{*} q$ *and* $h' \xrightarrow[\mathcal{R}]{*} h$.

Proof. (sketch) Let $s \in Q$ be such that $h \xrightarrow[\mathcal{A}']{*} s$ and let us call ρ this reduction. With a commutation of transitions, we can assume that ρ has the following form,

$$\rho : h = t_1 \ldots t_m \xrightarrow[\mathcal{A}']{*} \underbrace{s_1 \ldots s_m \xrightarrow[\mathcal{A}']{*} s}_{\rho_0}$$

where $t_1, \ldots, t_m \in \mathcal{T}(\Sigma)$, $s_1, \ldots, s_m \in Q$, and for all $1 \leq i \leq m$, $t_i \xrightarrow[\mathcal{A}']{*} s_i$, and the last step of this reduction involves a vertical transition $a(q^{a_1 \cdots a_n}) \to s_i$ or $b(q_{a_1 \ldots a_n}) \to s_i$. The proof is by induction on the length of ρ.

The shortest possible ρ has 2 steps: $h = t_1 = a(\varepsilon) \xrightarrow[\mathcal{A}_0]{} a(q^a) \xrightarrow[\mathcal{A}_0]{} q = s$ and (iii) holds immediately with $h' = h$, by Lemma 12.

For the induction step, we consider the length of ρ_0. If $|\rho_0| = 0$, we have necessarily $m = 1$, and the reduction ρ has one of the two following forms ($\boldsymbol{v} \in Q^*$).

$$h = t_1 = b(h') \xrightarrow[\mathcal{A}']{*} b(\boldsymbol{v}) \xrightarrow[\mathcal{A}']{*} b(q_{a_1 \ldots a_n}) \xrightarrow[\mathcal{A}']{} q_{a_1 \ldots a_n} = s_1 = s \quad (1)$$

$$h = t_1 = a_n(h') \xrightarrow[\mathcal{A}']{*} a_n(\boldsymbol{v}) \xrightarrow[\mathcal{A}']{*} a_n(q^{a_1 \cdots a_n}) \xrightarrow[\mathcal{A}_0]{} q_{a_1 \ldots a_n} = s_1 = s \quad (2)$$

In the case (1), assume that the vertical transition $b(q_{a_1 \ldots a_n}) \to q_{a_1 \ldots a_n}$ has been added to $\mathcal{A}'$ because $\mathcal{R}/\mathcal{A}$ contains a rule $a_n(x) \to b\big(a_n(x)\big)$. By induction hypothesis (i) applied to the sub-reduction $h' \xrightarrow[\mathcal{A}']{*} q_{a_1 \ldots a_n}$, there exists $h_1 \in \mathcal{H}(\Sigma)$ such that $a_1(h_1) \xrightarrow[\mathcal{B}]{*} q$, and $a_1(h_1) \xrightarrow[\mathcal{R}]{a_1 \ldots a_n} h'$. It follows in particular that there exists h_n such that $a_n(h_n) \xrightarrow[\mathcal{R}]{*} h'$, and using the above (ap) rewrite rule, $a_n(h_n) \xrightarrow[\mathcal{R}]{} b\big(a_n(h_n)\big) \xrightarrow[\mathcal{R}]{*} b(h') = h$. Therefore, $a_1(h_1) \xrightarrow[\mathcal{R}]{a_1 \ldots a_n} h$ and (i) holds for h and s.

In the case (2), by induction hypothesis (ii) applied to the sub-reduction $h' \xrightarrow[\mathcal{A}']{*} q^{a_1 \cdots a_n}$, there exists $h_1 \in \mathcal{H}(\Sigma)$ such that $h_1 \xrightarrow[\mathcal{B}]{*} q^{a_1}$, hence $a_1(h_1) \xrightarrow[\mathcal{B}]{*} q$, and $a_1(h_1) \xrightarrow[\mathcal{R}]{a_1 \ldots a_n} a_n(h') = h$. Therefore (i) holds for h and s.

Assume now that $|\rho_0| > 0$, and let us analyze the horizontal transition rule used in the last step of ρ_0. In order to comply with spaces restrictions, we will present only one significant case in this extended abstract (see [10] for the other cases).

Case (ac). The last step of ρ_0 uses $u\, q^{a_1 \cdots a_n}\, v \to q^{a_1 \cdots a_n}$ and this transition has been added to Δ' because $\mathcal{R}/\mathcal{A}$ contains a rule $a_n(x) \to a_n(u\, x\, v)$, with $u, v \in Q^*_{\mathcal{A}}$. In this case, the reduction ρ has the following form,

$$h = \ell\, h'\, r \xrightarrow[\mathcal{A}']{*} u\, q^{a_1 \cdots a_n}\, v \xrightarrow[\mathcal{A}']{} q^{a_1 \cdots a_n} = s \quad (3)$$

where $\ell \xrightarrow[\mathcal{A}']{*} u$, $h' \xrightarrow[\mathcal{A}']{*} q^{a_1 \cdots a_n}$, and $r \xrightarrow[\mathcal{A}']{*} v$. By induction hypothesis (ii) applied to $h' \xrightarrow[\mathcal{A}']{*} q^{a_1 \cdots a_n}$, there exists h_1 such that $h_1 \xrightarrow[\mathcal{B}]{*} q^{a_1}$ and $a_1(h_1) \xrightarrow[\mathcal{R}]{a_1 \ldots a_n}$

$a_n(h')$, and by induction hypothesis (iii) applied to $\ell \xrightarrow[\mathcal{A}']{*} u$ (resp. $r \xrightarrow[\mathcal{A}']{*} v$), and by Lemma 12, there exists $\ell' \in \mathcal{H}(\Sigma)$ (resp. $r' \in \mathcal{H}(\Sigma)$) such that $\ell' \xrightarrow[\mathcal{A}]{*} u$ (resp. $r' \xrightarrow[\mathcal{A}]{*} v$) and $\ell' \xrightarrow[\mathcal{R}]{*} \ell$ (resp. $r' \xrightarrow[\mathcal{R}]{*} r$). It follows that $a_n(h') \xrightarrow[\mathcal{R}]{} a_n(\ell' \, h' \, r') \xrightarrow[\mathcal{R}]{*} a_n(\ell \, h' \, r) = a_n(h)$. Hence $a_1(h_1) \xrightarrow[\mathcal{R}]{a_1 \ldots a_n} a_n(h)$ and (ii) holds for h and s. $\square$

Corollary 14. $L(\mathcal{A}') \subseteq \mathit{post}^*_{\mathcal{R}/\mathcal{A}}(L)$

Proof. By definition of Q^{f}, $h \in L(\mathcal{A}')$ iff $h \xrightarrow[\mathcal{A}']{*} q \in P^{\mathsf{f}} = Q^{\mathsf{f}}_L$, and $P^{\mathsf{f}} \subseteq P \setminus P_{\mathsf{in}}$. By Lemma 13, case (iii), it follows that $h \in \mathit{post}^*_{\mathcal{R}/\mathcal{A}}(L(\mathcal{B}, q)) \subseteq \mathit{post}^*_{\mathcal{R}/\mathcal{A}}(L)$. $\square$

Lemma 15 (Completeness). *For all $h \in \mathcal{H}(\Sigma)$ and $s \in Q$, if $h \xrightarrow[\mathcal{A}_0]{*} s$ and $h \xrightarrow[\mathcal{R}]{*} h'$, then $h' \xrightarrow[\mathcal{A}']{*} s$.*

The proof is by induction on the length of the rewrite sequence $h \xrightarrow[\mathcal{R}]{*} h'$ (see [10]). As another consequence of the result of [8] on the rewrite closure of HA languages under monadic HRS, the backward closure of a HA language under an uPHRS is HA.

The rules of type (ren), (as), (ap) and (rpl) can be easily simulated by the HRS of Theorem 11. In particular, the parameters' semantics can be simulated using ground rewrite rules (with such rules, a symbol can generate a HA language). The rules (ac) are not 1-childvar and the rules (del) is not inverse-monadic.

Example 8 shows the problems that can arise when combining in one single rewrite rule two rules of the form (as) and (ac), forcing synchronization of two updates. Note that the rule $a(x) \to c \, a(e \, x \, g) \, d$ of this example can be simulated by the 2 rules $a(x) \to c \, a'(x) \, d$ and $a'(x) \to a(e \, x \, g)$. The former rule is of the type of Theorem 11 (it combines types (as) and (ren)). The latter (which is not 1-varchild) combines types (ac) and (ren). This shows that such combinations can also lead to the behavior exposed in Example 8.

5 Future Works

As for future works on $\mathsf{CF}^2\mathsf{HA}$ languages several directions deserve to be followed. A first direction might be to derive pumping properties for these classes of languages. A second direction would be to look for an analogous of Parikh characterization for the number of different symbols occurring in the hedges of given $\mathsf{CF}^2\mathsf{HA}$ languages. One may define and study HRS with counting constraints on horizontal and vertical paths.

Finally, it is worth investigating the parallel rewriting of [14], on all a-positions, since it is closer to the semantics of XQUF, and trying to get an analogous of Theorem 11 for the parallel rewrite closure.

Acknowledgements. The authors wish to thank the reviewers and Giorgio Delzanno for interesting discussions on rewrite based models and verification techniques for XQUF.

References

1. 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)
2. Bouajjani, A., Touili, T.: On Computing Reachability Sets of Process Rewrite Systems. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 484–499. Springer, Heidelberg (2005)
3. Bravo, L., Cheney, J., Fundulaki, I.: ACCOn: checking consistency of XML write-access control policies. In: Proc. 11th Int. Conf. on Extending Database Technology (EDBT). ACM Int. Conf. Proc. Series, vol. 261, pp. 715–719. ACM (2008)
4. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), `http://tata.gforge.inria.fr/` (release October 12, 2007)
5. Coquide, J.L., Dauchet, M., Gilleron, R., Vagvolgyi, S.: Bottom-up tree pushdown automata: Classification and connection with rewrite systems. Theoretical Computer Science 127, 69–98 (1994)
6. Feuillade, G., Genet, T., Viet Triem Tong, V.: Reachability Analysis over Term Rewriting Systems. Journal of Automated Reasoning 33(3-4), 341–383 (2004)
7. Fundulaki, I., Maneth, S.: Formalizing XML access control for update operations. In: Proc. of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 169–174. ACM (2007)
8. Jacquemard, F., Rusinowitch, M.: Closure of Hedge-Automata Languages by Hedge Rewriting. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 157–171. Springer, Heidelberg (2008)
9. Jacquemard, F., Rusinowitch, M.: Rewrite-based verification of XML updates. In: Proc. of the 12th ACM SIGPLAN Int. Symp. on Principles and Practice of Declarative Programming (PPDP), pp. 119–130. ACM (2010)
10. Jacquemard, F., Rusinowitch, M.: Rewrite Closure and CF Hedge Automata (2012), `http://hal.inria.fr/hal-00752496` (long version)
11. Murata, M.: Hedge automata: a formal model for XML schemata (2000), `http://www.xml.gr.jp/relax/hedge_nice.html`
12. Ohsaki, H., Seki, H., Takai, T.: Recognizing Boolean Closed a-Tree Languages with Membership Conditional Rewriting Mechanism. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 483–498. Springer, Heidelberg (2003)
13. Salomaa, K.: Deterministic tree pushdown automata and monadic tree rewriting systems. J. Comput. Syst. Sci. 37(3), 367–394 (1988)
14. Solimando, A., Delzanno, G., Guerrini, G.: Automata-based Static analysis of XML Document Adaptation. In: Proceedings 3d International Symposium on Games, Automata, Logics and Formal Verification, vol. 96, pp. 85–98 (2012)
15. Touili, T.: Computing transitive closures of hedge transformations. In: Proc. 1st International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS). eWIC Series. British Computer Society (2007)

Linear-Time Version of Holub's Algorithm for Morphic Imprimitivity Testing

Tomasz Kociumaka[1], Jakub Radoszewski[1], Wojciech Rytter[1,2,⋆], and Tomasz Waleń[3,1,⋆⋆]

[1] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland
{kociumaka,jrad,rytter,walen}@mimuw.edu.pl

[2] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Chopina 12/18, 87-100 Toruń, Poland

[3] Laboratory of Bioinformatics and Protein Engineering, International Institute of Molecular and Cell Biology in Warsaw, Poland, Ks. Trojdena 4, 02-109 Warsaw, Poland

Abstract. Stepan Holub (Discr. Math., 2009) gave the first polynomial algorithm deciding whether a given word is a nontrivial fixed point of a morphism. His algorithm works in quadratic time for large alphabets. We improve the algorithm to work in linear time. Our improvement starts with a careful choice of a subset of rules used in Holub's algorithm that is necessary to grant correctness of the algorithm. Afterwards we show how to choose the order of applying the rules that allows to avoid unnecessary operations on sets. We obtain linear time using efficient data structures for implementation of the rules. Holub's algorithm maintains connected components of a graph corresponding to specially marked positions in a word. This graph is of quadratic size for large alphabet. In our algorithm only a linear number of edges of this conceptual graph is processed.

1 Introduction

We consider finite words that are fixed points of morphisms. A word $w \in \Sigma^*$ is called a fixed point of a morphism $h : \Sigma^* \to \Sigma^*$ if $h(w) = w$. We say that w is a *trivial* fixed point of h if $h(a) = a$ for every letter actually occurring in w. The problem of finding all fixed points of a morphism was already studied in [5,6,11]. A more difficult problem of finding for a given word $w \in \Sigma^*$ a morphism h such that w is a non-trivial fixed point of h was first studied by Holub [8,7]. The problem turned out to have a polynomial-time solution although a similar problem of finding for a given pair of words w, v a morphism such that $h(w) = v$ was shown to be NP-complete [2]. A word w is called *morphically imprimitive* if there exists a morphism h such that w is a non-trivial fixed point of h. Otherwise, w is called *morphically primitive*.

⋆ The author is supported by grant no. N206 566740 of the National Science Centre.

⋆⋆ The author is supported by Iuventus Plus grant (IP2011 058671) of the Polish Ministry of Science and Higher Education.

Example 1. The word $w = aeecbdebaabdebeec$ is morphically imprimitive; it is a non-trivial fixed point of the following morphism h:

$$h: \ a \to a, \quad b \to \varepsilon, \quad c \to eec, \quad d \to bdeb, \quad e \to \varepsilon.$$

On the other hand, the word $w' = aeecbdebaabdebec$ is morphically primitive.

Let w be a word of length n over an alphabet Σ of size m. We assume that Σ is an *integer* alphabet, i.e. each letter $a \in \Sigma$ has a unique integer identifier of magnitude $n^{O(1)}$. The **main problem** considered here is to decide, for the given word w, whether it is morphically imprimitive and, if so, find a morphism h such that w is a non-trivial fixed point of h.

Holub [8] presented an $O((m + \log n) \cdot n)$ time algorithm for this problem and more recently Matocha and Holub [10] slightly improved the time complexity to $O(m \cdot n)$. Both solutions are potentially quadratic if $m = \Theta(n)$. We give an algorithm which works in $O(n)$ time for arbitrary integer alphabet.

The main components of Holub's algorithm are two sets $L, R \subseteq \{0, \ldots, n\}$. The algorithm performs implicitly quadratically many insertions into those sets in worst case. Each insertion can be either useless, when the inserted element is already in the set, or useful, otherwise. The crucial improvements to the Holub's algorithm presented in this paper are:

1. Reduction of the number of insertions into L to $O(n)$, this is obtained by changing the logics of basic operations performed in Holub's algorithm.
2. Reduction of the number of useless insertions into R to $O(n)$ using efficient data structures.

2 Preliminaries

By $Occ(a, w)$ (or simply $Occ(a)$) we denote the set of positions in w where the letter a occurs. We also denote $|w|_a = |Occ(a, w)|$. The set of letters actually occurring in w is denoted by $alph(w)$.

Let $F = (w_1, \ldots, w_k)$ be a factorization of the word w, $w = w_1 \ldots w_k$. We say that $e \in alph(w)$ is a *key-letter* if for each $i, j \in \{1, \ldots, k\}$ we have

$$|w_i|_e \leq 1 \quad \text{and} \quad (|w_i|_e = |w_j|_e = 1 \text{ implies that } w_i = w_j).$$

In other words, each factor contains exactly one occurrence of a key-letter and factors are determined by the key-letters.

If e is a key-letter, let $F_e = w_i$, where $|w_i|_e = 1$. We say that e is a *standard* key-letter if no key-letter occurs in F_e before the occurrence of e.

We say that F is a *morphic* factorization if each factor w_i contains a key-letter. Let $\mathcal{E}$ be the set of standard key-letters of a morphic factorization F. Then setting $h(e) = F_e$ for $e \in \mathcal{E}$ and $h(a) = \varepsilon$ for $a \notin \mathcal{E}$ yields a morphism h such that $h(w) = w$. A morphism obtained this way is called a *standard* morphism of w. The elements of $\mathcal{E}$ are called *expanding* letters and the elements of $alph(w) \setminus \mathcal{E}$ are called *mortal* letters. Holub [8] proved the following equivalent condition on when a word is morphically imprimitive:

Lemma 2 (Theorem 1 in [8]). *A word w is morphically imprimitive if and only if it has a morphic factorization F which is nontrivial, that is, has less than $|w|$ factors.*

Let $w[i]$ denote the i-th letter of w (for $1 \le i \le n$) and $w[i,j]$ denote the word $w[i]w[i+1]\dots w[j]$. Let $\{0,1,\dots,n\}$ be the set of *inter-positions* of w located in the beginning of the word ($i=0$), between any two letters of the word ($1 \le i < n$) or at the end of the word ($i=n$).

For a letter $a \in alph(w)$, we define the *right range* of a as a word $\mathbf{r}_a$ such that $a\mathbf{r}_a$ is the longest common prefix of all suffixes of w starting with a, and similarly the *left range* $\mathbf{l}_a$ such that $\mathbf{l}_a a$ is the longest common suffix of all prefixes of w ending with a. We denote $\mathbf{n}_a = \mathbf{l}_a a \mathbf{r}_a$. Let $r_a = |\mathbf{r}_a|$, $l_a = |\mathbf{l}_a|$ and $n_a = |\mathbf{n}_a|$.

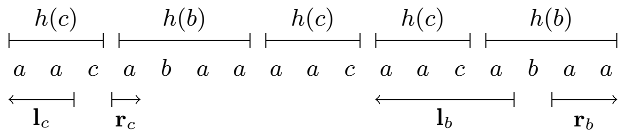


Fig. 1. For this word $\mathbf{n}_a = a$, $\mathbf{n}_b = aacabaa$ and $\mathbf{n}_c = aaca$. This word is morphically imprimitive due to the morphism $h(a)=\varepsilon$, $h(b)=abaa$, $h(c)=aac$.

Denote by $\prec$ an ordering of $alph(w)$ according to $|w|_a$: $a \prec b$ if and only if $|w|_a < |w|_b$. For any $0 \le i < j \le n$, define $\alpha(i,j) = \min(alph(w[i+1,j]))$, where the minimum is taken according to the order $\prec$. If there are multiple equal letters, we choose the one with the leftmost first occurrence within $w[i,j]$.

Holub's algorithm for testing morphic imprimitivity can be stated as follows. The paper [8] provides a simple algorithm that recovers the morphism from the sets E, L, R.

Algorithm VERSION1: Holub's algorithm.

```
Maintain a triple of sets $(E,L,R)$, initially equal to $(\emptyset,\emptyset,\emptyset)$, and use the
following set of rules (a)-(e) to extend these sets until $E = alph(w)$ or
none of the actions alters the triple:

(a) $L := L \cup \{0,n\}$; $R := R \cup \{0,n\}$
(b) **if** $w[i] \in E$ **then** $L := L \cup \{i-1\}$; $R := R \cup \{i\}$
(c) **if** $w[i,j] = \mathbf{n}_a$ *for some* $a \in E$ **then** $R := R \cup \{i-1\}$; $L := L \cup \{j\}$
(d) **if** $w[i,j] = w[i',j'] = \mathbf{n}_a$ *for some* $a \in E$ **then**
      – **if** $i+k \in R$ *for some* $-1 \le k \le j-i$ **then** $R := R \cup \{i'+k\}$
      – **if** $i+k \in L$ *for some* $-1 \le k \le j-i$ **then** $L := L \cup \{i'+k\}$
(e) **if** $i < j$ **and** $i \in L$ **and** $j \in R$ **then** $E := E \cup \{\alpha(i,j)\}$

**if** $E \neq alph(w)$ **then return** *true* **else return** *false*
```

Let h be a standard morphism of w. Let $\mathcal{E}_h$ denote the set of expanding letters of h. Moreover, let us define two subsets of inter-positions, $\mathcal{L}_h$ of *left* inter-positions and $\mathcal{R}_h$ of *right* inter-positions:

$$\mathcal{L}_h = \{i : |h(w[1,i])| \le i\}, \qquad \mathcal{R}_h = \{i : |h(w[1,i])| \ge i\}.$$

Definition 3. *A triple (E, L, R) is called* correct *if $E \subseteq \mathcal{E}_h$, $L \subseteq \mathcal{L}_h$ and $R \subseteq \mathcal{R}_h$ for any standard morphism h.*

Lemmas 4, 6, 7 and 8 of [8] can be stated succinctly as the following fact:

Fact 4. *Extending a correct triple using any of the rules (a)-(e) leads to a correct triple. In particular, if any sequence of actions corresponding to (a)-(e) leads to $E = alph(w)$ then w is morphically primitive.*

3 The Structure of Our Algorithm

Now we describe the set of rules used in our algorithm, which is a subset of rules from Holub's algorithm, and show that this set suffices to construct an algorithm for testing morphic primitivity (see the following Lemma 6 and Theorem 7).

Let us introduce some notation. Let A be a set of integers. We define the predecessor in A and successor in A in a standard way:

$$succ_A(x) = \min\{y : y \in A, y > x\}, \quad pred_A(x) = \max\{y : y \in A, y < x\}.$$

We assume $\min \emptyset = \infty$, $\max \emptyset = -\infty$.

Let $Occ(E) = \{i : w[i] \in E\}$ be the set of occurrences of expanding letters in w, we call them *expanding occurrences*. We slightly abuse the notation and write $succ_E$ and $pred_E$ instead of $succ_{Occ(E)}$ and $pred_{Occ(E)}$.

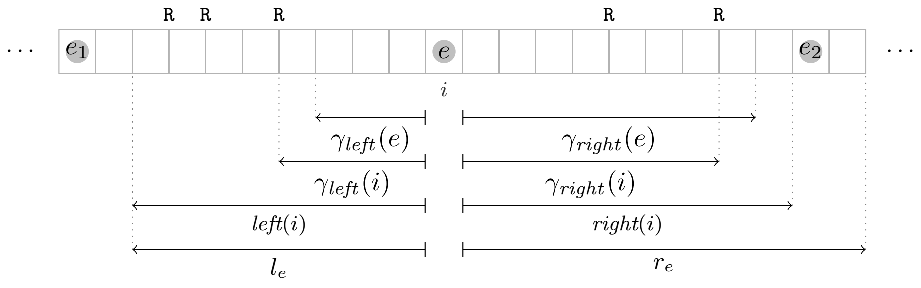


Fig. 2. Illustration of the main notions of the algorithm.

The bottleneck of Holub's algorithm is performing the rule (d). Our crucial improvement, which makes linear time implementation possible, is twofold. First, we shrink the ranges so that they never exceed other expanding positions. This way, each inter-position lies in at most two ranges — originating at the closest expanding positions — and consequently propagation is simpler to control. We introduce *left* and *right* functions, defined for $i \in Occ(E)$:

$$left(i) = \min(l_{w[i]},\, i - pred_E(i) - 1), \quad right(i) = \min(r_{w[i]},\, succ_E(i) - i - 1).$$

Secondly, we only propagate elements of R from selected inter-positions. For $i \in Occ(E)$, define:

$$\gamma_{left}(i) = i - pred_R(i) - 1, \quad \gamma_{right}(i) = pred_R(i + right(i) + 1) - i.$$

Less formally, $\gamma_{left}(i)$ shows the location of the rightmost element of R before the position i, while $\gamma_{right}(i)$ points to the rightmost R within the range of the position i (see Fig. 2). We extend these definitions to expanding letters:

$$\gamma_{left}(e) = \min\{\gamma_{left}(i) : i \in Occ(e)\}, \quad \gamma_{right}(e) = \max\{\gamma_{right}(i) : i \in Occ(e)\}.$$

Thus we arrive at the following Algorithm Version2.

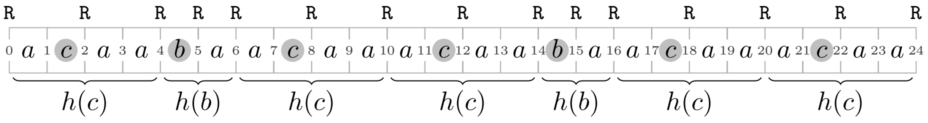

R
0 a 1 c 2 a 3 a 4 b 5 a 6 a 7 c 8 a 9 a 10 a 11 c 12 a 13 a 14 b 15 a 16 a 17 c 18 a 19 a 20 a 21 c 22 a 23 a 24
h(c)
h(b)

Fig. 3. In the above word $\mathbf{n}_a = a$, $\mathbf{n}_b = acaabaacaaacaa$ and $\mathbf{n}_c = acaa$. Let $E = \{b, c\}$, these letters are marked in the figure. We have $\gamma_{left}(b) = 0$, $\gamma_{right}(b) = 1$, $\gamma_{left}(c) = 1$, $\gamma_{right}(c) = 2$. The morphic factorization is obtained by cutting the word at inter-positions $\{i - 1 - \gamma_{left}(w[i]) : i \in Occ(E)\}$; thus $h(b) = ba$, $h(c) = acaa$, $h(a) = \varepsilon$.

Algorithm Version2

```
Perform, in any order, the following operations on a triple of sets
$(E, L, R)$, initially equal to $(\emptyset, \emptyset, \emptyset)$, until none of the operations alters the
triple (note that the operations (b'), (c'), (d'), (e') together perform only
a subset of actions from operations (b), (c), (d), (e) in Holub's algorithm):

 (a)  $L := L \cup \{0, n\}$; $R := R \cup \{0, n\}$
(b')  if $w[i] \in E$ then $R := R \cup \{i\}$
(c')  if $w[i] \in E$ then $R := R \cup \{i - left(i) - 1\}$; $L := L \cup \{i + right(i)\}$
(d')  if $w[i] \in E$ then $R := R \cup \{i - 1 - \gamma_{left}(w[i]), i + \gamma_{right}(w[i])\}$
(e')  if $i < j$ and $succ_R(i) = j$ and $pred_L(j) = i$ and
      $\{w[i+1], \ldots, w[j]\} \cap E = \emptyset$ then
          $E := E \cup \{\alpha(i, j)\}$

if $E \neq alph(w)$ then return true else return false
```

In our Algorithm Version2 we use rules weaker than Holub's, hence they satisfy the same property as in Fact 4:

Fact 5. *Extending a correct triple using any of the rules (a), (b'), (c'), (d'), (e') leads to a correct triple. In particular, if any sequence of actions corresponding to these rules leads to $E = alph(w)$ then w is morphically primitive.*

Lemma 6. *Assume the execution of Algorithm* Version2 *has finished with the triple (E, L, R). Then for each $1 \le i < j \le n$ such that $w[i] \in E$ and $j = succ_E(i)$ we have*

$$i + \gamma_{right}(w[i]) = j - 1 - \gamma_{left}(w[j]). \tag{1}$$

Proof. First note that, under the conditions of the lemma, both $\gamma_{right}(w[i])$ and $\gamma_{left}(w[j])$ are finite. Indeed, the former is finite since $i \in R$ and the latter is finite since the left end of the range of j, that is, $j - 1 - left(j)$, belongs to R.

For a proof by contradiction assume that (1) does not hold for some i, j. Assume first that $i + \gamma_{right}(w[i]) < j - 1 - \gamma_{left}(w[j])$. We have two cases here. Let $u = i + right(i)$ and $v = j - 1 - \gamma_{left}(w[j])$. If $u < v$ then one could perform the operation (e) on $u \in L$ and $v \in R$. However, by taking $u' = pred_L(v)$ and $v' = succ_R(u')$ we obtain a pair of positions $u', v' \in [u, v]$ that would enable us to perform the (e') operation, hence the algorithm has not finished yet. In the opposite case, if $v \le u$, the inter-position $v \in R$ would contradict the definition of $\gamma_{right}(w[i])$.

Now assume that $i + \gamma_{right}(w[i]) > j - 1 - \gamma_{left}(w[j])$. Then the inter-position $i + \gamma_{right}(w[i]) \in R$ contradicts the definition of $\gamma_{left}(w[j])$. □

Theorem 7. *Algorithm* Version2 *correctly decides whether w is morphically imprimitive. If so, the corresponding morphic factorization of w is described by the following factors (see Fig. 3):*

$$\{w[i - \gamma_{left}(w[i]), i + \gamma_{right}(w[i])] \,:\, w[i] \in E\}. \tag{2}$$

Proof. Assume that the triple (E, L, R) returned by our algorithm satisfies $|E| < |alph(w)|$. Then, by Lemma 6, the factorization (2) is a nontrivial morphic factorization of w that induces a morphism h such that $h(w) = w$.

Now assume that $|E| = |alph(w)|$ in the final triple of the algorithm. Then, by Fact 5, the word w is morphically primitive. □

4 Efficient Version of the Algorithm

Algorithm Version2 is of a nondeterministic nature, inserting new elements to L, R and E is performed in any order until the configuration stabilizes. Now we proceed to efficient deterministic implementation, which is *neighbour-driven*: the actions are performed locally between neighbours, new elements inserted to R affect their *E-neighbours* which potentially generate new elements of R in *neighbourhoods* (ranges) of occurrences of these E-neighbours. For a position i its left/right E-neighbour is at position $pred_E(i)/succ_E(i)$.

We say that an interval $[i, j]$ is *loose* if the rule (e') from Algorithm Version2 can be applied to (i, j). In other words, an interval is loose if it can generate a new expanding letter $\alpha(i, j)$.

Description of One Iteration of the Algorithm. In one main iteration Algorithm Version3 adds a single new expanding letter e (the letter is generated using the function *ProcessInterval*). We maintain the set of *loose* intervals, which are potential generators of such letters, in *IntervalsSet* data structure. From now only new inter-positions are inserted to R and L until the configuration stabilizes. This phase corresponds to multiple application of the (most expensive) part (d') in Algorithm Version2. A new expanding letter e, by the rules (b') and (c'), generates a number of new elements of R and L. New elements of R by a *chain-reaction* cause further insertions to R that are related to other letters e', then they affect others and so on. This chain-reaction is performed using the queue *LettersQueue*. Each insertion of a new element to R can affect all occurrences of its both E-neighbours (due to the rule (d')); we insert these E-neighbours into *LettersQueue* to be processed later. We successively dequeue a letter e from *LettersQueue* and compute the set of new elements of R that occur due to the rule (d') applied for any $w[i] = e$. Each new insertion into R is processed by the function *Propagate*(k). It is responsible for insertion into *LettersQueue* of the E-neighbours of k and updating their γ-values.

We proceed now to a more detailed description of the whole algorithm. The algorithm starts with $L = R = \{0, n\}$, $E = \emptyset$ and when it terminates none of the operations (b'), (c'), (d'), (e') can be performed. For each $e \in E$ the values $\gamma_{left}(e)$ and $\gamma_{right}(e)$ are explicitly stored, whereas the values $\gamma_{left}(i)$, $\gamma_{right}(i)$, $left(i)$ and $right(i)$ for $i \in Occ(E)$ are computed on demand using predecessor/successor queries.

The following function *ProcessInterval*(i, j) performs the action (e'), and then the actions (b'), (c') from Algorithm Version2. The function also accounts for the situation when the new expanding letter causes the right range of its E-neighbour to decrease. It returns the set of newly inserted elements of R.

```
**Function** ProcessInterval$(i, j)$

  $e := \alpha(i, j)$; $E := E \cup \{e\}$; $NewR := \emptyset$
  **foreach** $p \in Occ(e)$ **do**
      $NewR := NewR \cup \{p, p - left(p) - 1\}$
      $L := L \cup \{p + right(p)\}$
  $NewR := NewR \setminus R$; $R := R \cup NewR$
  compute $\gamma_{right}(e)$, $\gamma_{left}(e)$; add $e$ to $LettersQueue$
  **foreach** $p \in Occ(e)$ **do**
      $e' := w[pred_E(p)]$; $\{e'$ is $E$-neighbour of position $p\}$
      **if** $\gamma_{right}(e') > right(pred_E(p))$ **then**
          $\gamma_{right}(e') := \max\{pred_R(p' + right(p') + 1) - p' \ : \ p' \in Occ(e')\}$
  **return** $NewR$
```

The function *Propagate* is called each time a new element is inserted to R. It is responsible for inserting into *LettersQueue* and updating the γ-values of expanding letters.

```
Function Propagate($i$)
    {we assume that $i \in R$}
    $e_1 := w[pred_E(i+1)];\ e_2 := w[succ_E(i)]$
    {$e_1, e_1$ are $E$-neighbours of inter-position $i$}
    add $e_1, e_2$ to $LettersQueue$
    update $\gamma_{right}(e_1)$ and $\gamma_{left}(e_2)$ (if necessary)
```

For each letter $e \in E$, we store the set of its occurrences for which the rule (d') would insert a new inter-position to R:

$$ActiveSet(e) = \ \{i \in Occ(e) \ : \gamma_{right}(i) < \gamma_{right}(e) \ \vee \ \gamma_{left}(i) > \gamma_{left}(e)\}.$$

The implementation of operations required to store these sets is provided in the next section. We additionally need a queue *LettersQueue* that stores elements of E. If any expanding letter e satisfies $ActiveSet(e) \neq \emptyset$ then it is guaranteed to be present in the queue.

```
Algorithm Version3
    $L := R := \{0, n\};\ E := \emptyset$
    $IntervalsSet := \{[0, n]\}$
    while $IntervalsSet$ not empty do {Main Iteration}
        let $[i, j]$ be any element of $IntervalsSet$
        $NewR :=$ ProcessInterval($i, j$)
        {New expanding letter $\alpha(i, j)$ has been added to $E$}
        foreach $k \in NewR$ do Propagate($k$)
        while $LettersQueue$ not empty do
            $e := dequeue(LettersQueue)$
            foreach $i \in ActiveSet(e)$ do
                $NewR := \{i + \gamma_{right}(e), i - 1 - \gamma_{left}(e)\} \setminus R$
                $R := R \cup NewR$
                foreach $k \in NewR$ do $Propagate(k)$
    if $E \neq alph(w)$ then return $true$ else return $false$
```

Theorem 8. [Main Result]
The problem of morphic imprimitivity can be solved in linear time.

Proof. Correctness of Algorithm Version3 follows from correctness of Algorithm Version2. It suffices to note that at the end of execution none of the rules (a), (b'), (c'), (d'), (e') can be applied. Clearly, we apply all the rules (a), (b'), (c') as soon as possible. As for the rules (d') and (e'), if any of the rules can be applied, *LettersQueue* and *IntervalsSet* is non-empty, respectively.

Let us investigate the time complexity of Algorithm Version3. In the next section we show that, after $O(n)$ preprocessing, predecessor/successor queries for L, R and E can be answered in $O(1)$ time, *ActiveSet*-queries can be answered in time proportional to the number of returned positions (plus one) and *IntervalsSet* can be maintained to answer any-element-removal queries in $O(1)$ time. Moreover, we show there that the ranges l_a, r_a can be precomputed in $O(n)$ time and the $\alpha(i, j)$ queries can be answered in $O(1)$ time.

The main point, with respect to the time complexity, is that the work is amortized by the number of occurrences of a newly inserted letter (this concerns ProcessInterval calls and the steps of the main while-loop) and by the number of newly inserted elements of R (this concerns Propagate calls and all the remaining loops of the main algorithm). This yields $O(n)$ time. □

5 Auxiliary Data Structures

In this section we show how to implement the data structures which support computing letter ranges and $\alpha(i, j)$, predecessor/successor queries and efficiently maintain the sets *IntervalsSet* and *ActiveSet*(e) for each $e \in E$ in the main algorithm.

5.1 Applications of RMQ and LCP

We use two well-known data structures with $O(n)$ preprocessing time and $O(1)$ query time.

Range Minimum Queries (RMQ). We are given an array $A[1..n]$ of integers. This array is preprocessed to answer the following queries: for an interval $[i, j]$ (where $1 \le i \le j \le n$), find any $k_0 \in [i, j]$ such that $A[k_0] = \min\{A[k] : k \in [i, j]\}$. The best known RMQ data structures have $O(n)$ preprocessing time and $O(1)$ query time [4].

Longest Common Prefix Queries (LCP). Let w be a word of length n. For a pair of indices i, j, $1 \le i, j \le n$, we introduce $lcp(i, j)$ as the length of the longest common prefix between $w[i..n]$ and $w[j..n]$. The *lcp* queries can be performed in $O(1)$ time after $O(n)$ time preprocessing [1,9].

The RMQ data structure allows efficient computing of $\alpha(i, j)$.

Lemma 9. *$\alpha(i, j)$ can be computed in $O(1)$ time after $O(n)$ preprocessing.*

The following lemma follows easily by multiple application of *lcp*-queries to compute r_a, l_a for all $a \in alph(w)$.

Lemma 10. *The values r_a, l_a for all $a \in alph(w)$ can be computed in $O(n)$ time.*

5.2 Answering Incremental Predecessor/Successor Queries

As a tool we use the following known problem.

Marked Ancestor Problem: Let T be a rooted tree with n nodes, each of which can be in two states: marked or unmarked. We are to process a sequence of m operations of the following two types:

mark(v): mark node v;

firstmarked(v): return the first marked node on the path from v to the root.

This classical problem can be solved on-line in $O(n + m)$ time, see [3]. This implies an efficient algorithm for answering incremental predecessor/successor queries.

Lemma 11. *A sequence of $O(n)$ predecessor/successor queries for the sets L, R and $Occ(E)$ can be handled in $O(n)$ total time.*

Proof. Note that each of the sets L, R, $Occ(E)$ is a subset of $\{0, \dots, n\}$ (initially empty) and the only operation performed on these sets is insertion. Hence, a sequence of m predecessor queries on these sets can be performed on-line in $O(n+m)$ time in total with the data structure for the Marked Ancestor Problem. We use a tree T that is a single path of $n + 1$ nodes, insertion to the sets corresponds to the operation *mark* and *firstmarked* returns the predecessor of a node. Successor queries are handled analogously. $\square$

5.3 Computation of ActiveSets

All the operations performed on the respective ActiveSets are provided by the following auxiliary data structure for dynamic storage of maxima.

Decremental-Maxima: We store an array $t[1\,.\,.\,m]$ of integers, initially $t[i] = -\infty$ for all i. We support the following types of operations:

increasevalue(i, v): set $t[i] := \max(v, t[i])$;

max$()$: return $\max\{t[i] : i = 1, \dots, m\}$;

notmaximal$()$: return any i such that $t[i] \neq max()$ or **nil** if no such i exists;

reset$()$: set $t[i] = -\infty$ for all $i = 1, \dots, m$.

Lemma 12. *The Decremental-Maxima data structure can be initialized in $O(m)$ time so that reset$()$ operation is performed in $O(m)$ time and every other operation is performed in $O(1)$ time.*

Proof. We store the array t, the current maximum M and two lists L and L' of elements from $\{1, \dots, m\}$. Each element belongs to exactly one of the lists and as $p[i]$ we store the pointer to its current location in its list. The list L contains all i for which $t[i] = M$ and L' contains the remaining elements of the domain.

The *max*$()$ operation is performed by returning M. In *increasevalue*(i, v) we update $t[i]$ if $v > t[i]$. If $v = M$ then i is moved from L' to L and if $v > M$ then v becomes M, the list L is appended to L' and L becomes the single element i. Finally, *notmaximal*$()$ returns any element of the list L' if it is non-empty. $\square$

Lemma 13. *All the operations on all ActiveSets can be implemented in $O(n)$ total time.*

Proof. For a given $e \in E$, all the elements $i \in \mathit{ActiveSet}(e)$ can be divided into those for which $\gamma_{\mathit{right}}(i) > \gamma_{\mathit{right}}(e)$ and those for which $\gamma_{\mathit{left}}(i) < \gamma_{\mathit{left}}(e)$. We handle both cases separately. Now we show how to solve the former case with a single instance of the Decremental-Maxima data structure with $m = |\mathit{Occ}(e)|$.

When a new letter e is inserted to E (in ProcessInterval(i, j)), we build the data structure corresponding to e: we apply *reset* and *increasevalue* for all $p \in \mathit{Occ}(e)$ using pred_R queries for the right endpoints of the ranges. This takes $O(|\mathit{Occ}(e)|)$ time for each new letter, $O(n)$ time in total.

Updates of the data structure take place when $\gamma_{\mathit{right}}(p)$ changes for any $p \in \mathit{Occ}(e)$. Whenever a new element i is inserted to R, *increasevalue* may be called only for $\mathit{pred}_E(i)$ (if i is in its range). In total we have $O(n)$ such calls.

Throughout the algorithm γ_{right} may occasionally decrease. This takes place in the last for-all-loop of ProcessInterval(i, j), due to insertion of the new letter e, $\gamma_{\mathit{right}}(e')$ may decrease. In this case we recompute the data structure for e' from scratch in $O(|\mathit{Occ}(e')|)$ time. Let $e'_1, \dots, e'_l$ be all the letters for which γ_{right} decreased because of the new letter e. Note that each occurrence of each e'_i is a pred_E element of the corresponding occurrence of e. Hence

$$\sum_{i=1}^{l} |\mathit{Occ}(e'_i)| \;\leq\; |\mathit{Occ}(e)|.$$

Consequently, this yields a cost of $O(|\mathit{Occ}(e)|)$ per a new letter which gives $O(n)$ time in total.

The only remaining operation on ActiveSets is the "**foreach** $i \in \mathit{ActiveSet}(e)$" loop in the main algorithm. The total number of steps of this loop is $O(n)$ — since each step inserts a new element to R — and selecting any $i \in \mathit{ActiveSet}(e)$ is performed in $O(1)$ time using the *notmaximal* operation.

The case of γ_{left} is similar (we take minimum instead of maximum, also recomputation from scratch is never needed). In conclusion, all the necessary updates and queries on ActiveSets take $O(n)$ total time. □

5.4 Implementation of IntervalsSet

IntervalsSet is needed to generate new expanding letters. It is maintained as a linked list of pairs of integers.

Lemma 14. *All operations on IntervalsSet can be implemented in $O(n)$ total time.*

Proof. We implement the *IntervalsSet* as a linked list of loose intervals. We also store an array that, for each element $i \in \{0, \dots, n\}$, points to the loose interval in the list that has i as its endpoint (if there is any such loose interval).

At the beginning of the algorithm the list contains a single element $[0, n]$. Each insert operation on sets L and R causes at most one insertion to and at most one deletion from the *IntervalsSet*, the new loose interval is computed using a single predecessor/successor query. Similarly, each insertion of $e \in E$ causes at most one deletion from the *IntervalsSet* per each $p \in Occ(e)$, that is, a deletion of the formerly loose interval.

Consequently, we maintain up-to-date contents of the *IntervalsSet* after each insertion to sets L, R and $Occ(E)$ with $O(1)$-time overhead. □

6 Conclusions

We presented a linear time algorithm for deciding if a word is morphically imprimitive. We started from the original quadratic algorithm by Holub (Algorithm Version1), transformed it by reducing the set of rules used by the algorithm (Algorithm Version2) and finally provided several efficient data structures that enabled linear-time implementation of the previous version if a specific order of performing the rules is applied (Algorithm Version3). Algorithm Version3 tests if a word is morphically imprimitive. It can also provide a morphic factorization (due to Theorem 7).

References

1. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
2. Ehrenfeucht, A., Rozenberg, G.: Finding a homomorphism between two words is NP-complete. Inf. Process. Lett. 9(2), 86–88 (1979)
3. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. In: Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC), pp. 246–251 (1983)
4. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
5. Head, T.: Fixed languages and the adult languages of OL schemes. International Journal of Computer Mathematics 10(2), 103–107 (1981)
6. Head, T., Lando, B.: Fixed and stationary ω-words and ω-languages. In: Rozenberg, G., Salomaa, A. (eds.) The Book of L, pp. 147–156. Springer (1986)
7. Holub, S.: Algorithm for fixed points of morphisms — visualization, `http://www.karlin.mff.cuni.cz/~holub/soubory/Vizual/stranka2.html`
8. Holub, S.: Polynomial-time algorithm for fixed points of nontrivial morphisms. Discrete Mathematics 309(16), 5069–5076 (2009)
9. Ilie, L., Tinta, L.: Practical Algorithms for the Longest Common Extension Problem. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 302–309. Springer, Heidelberg (2009)
10. Matocha, V., Holub, S.: Complexity of testing morphic primitivity. CoRR abs/1207.5690v1 (2012)
11. Reidenbach, D., Schneider, J.C.: Morphically primitive words. Theor. Comput. Sci. 410(21-23), 2148–2161 (2009), `http://dx.doi.org/10.1016/j.tcs.2009.01.020`

From Regular Tree Expression to Position Tree Automaton[*]

Éric Laugerotte, Nadia Ouali Sebti, and Djelloul Ziadi

Laboratoire LITIS - EA 4108 Université de Rouen, Avenue de l'Université
76801 Saint-Étienne-du-Rouvray Cedex, France
{Eric.Laugerotte,Nadia.Ouali-Sebti,Djelloul.Ziadi}@univ-rouen.fr

Abstract. The word position automaton was introduced by Glushkov and McNaughton in the early 1960. This automaton is homogeneous and has $(||\mathrm{E}||+1)$ states for an expression of alphabetic width $||\mathrm{E}||$. In this paper this type of automata is extended to regular tree expressions and it is shown that the conversion of a regular tree expression of size $|\mathrm{E}|$ and alphabetic width $||\mathrm{E}||$ into its reduced tree automaton can be done in $O(|\mathrm{E}|\cdot||\mathrm{E}||)$ time. The time complexity of the algorithm proposed by Dietrich Kuske and Ingmar Meinecke is also proved in order to reach an $O(||\mathrm{E}||\cdot|\mathrm{E}|)$ time complexity for the problem of the construction of the equation automaton from a regular tree expression.

1 Introduction

In the case of words, it is agreed that each regular expression can be transformed into a non-deterministic finite automaton. Computer scientists have been interested in designing efficient algorithms for the computation of the position automaton. First of all, three well-known algorithms for computing this automaton exist. The first makes use of the notion of star normal form [2] of a regular expression. The second is based on a lazy computation technique [3]. The third is built on the so-called ZPC-structure [10]. The complexity of these three algorithms is quadratic w.r.t the size of the regular expression.

Finite Tree Automata constitute an efficient data structure used in various application areas such as logic, rewriting, linguistic, verification [9], XML,...

This study is motivated by the development of a library of functions for handling rational kernels [5] in the case of trees. The first problem consists in converting a regular tree expression into a tree transducer (automaton).

Recently Dietrich and Meinecke [7] proposed for a regular tree expression E an $O(\mathrm{R}\cdot|\mathrm{E}|^2)$ time Algorithm where $|\mathrm{E}|$ is the size of E and R is the maximal rank appearing in the ranked alphabet, to construct its equation automaton [8], [1]. This algorithm is an adaptation to trees of the one given by Champarnaud and Ziadi in the case of words [10]. This generalization is interesting although the adaptation of the word algorithm to trees is not obvious at all. Indeed the Champarnaud and Ziadi Algorithm, for the construction of the set of transitions,

[*] D. Ziadi was supported by the MESRS - Algeria under Project 8/U03/7015.

is based on the computation of some function called "Follow" which is not yet defined on trees. The complexity of $O(\mathrm{R}\cdot|\,\mathrm{E}\,|^2)$ is proved in this paper. For this reason the definition of this function in the case of trees is given in this paper while an efficient algorithm for its computation is proposed. Therefore, in one hand we prove the time complexity of the algorithm proposed by Dietrich and Meinecke [7]. In the other hand we give an efficient algorithm to compute position tree automaton.

The paper is organized as follows: Section 2 outlines finite tree automata and regular tree expressions. Next the notion of linearized regular tree expressions is given which allows the set of positions and the operators c−product list and c−closure list to be defined. The functions First and Follow which are a generalization of the ones introduced by Glushkov [6] are then defined. Thus the definition of position tree automaton and its reduced version associated to the regular tree expression is obtained. Afterwards, the main steps of the computation of the Follow function is described. Finally, the different results described in this paper are given in the conclusion.

2 Preliminaries

In this section the notions of tree languages, Finite Tree Automata (FTA), c−product and c−closure will be reviewed. Let (Σ, ar) be *a ranked alphabet*, where Σ is a finite set and ar represents the arity of Σ which is a mapping from Σ into $\mathbb{N}$. The set of letters of arity n is denoted by Σ_n. The elements of arity 0 are called constants. We denote by T_Σ the set of trees over Σ respecting the arity of each symbol. *A tree language* is a subset of T_Σ.
Let $\Sigma_{\geq 1} = \bigcup_{m\geq 1} \Sigma_m = \Sigma \backslash \Sigma_0$ denote the set of non-constant symbols from the ranked alphabet Σ.
A Finite Tree Automaton (FTA) [7,4] $\mathcal{A}$ is the tuple (Q, Σ, Q_T, Δ) where Q is the finite set of states, $Q_T \subseteq Q$ is the set of final states and $\Delta \subseteq (Q \times \Sigma_0) \cup \cup_{n\geq 1}(Q \times \Sigma_n \times Q_n)$ is the set of transition rules. A tree constant c is accepted by $\mathcal{A}$ if and only if there exists a transition $(q, c) \in \Delta$ with $q \in Q_T$. A tree $t = f(t_1, \ldots, t_n), n \geq 1$ is accepted by $\mathcal{A}$ iff there exists a transition $(q, f, q_1, \ldots, q_n) \in \Delta$ such that $q \in Q_T$ and for $1 \leq i \leq n$, the tree t_i is accepted by FTA $(Q, \Sigma, \{q_i\}, \Delta)$.
The tree language $\mathcal{L}(\mathcal{A})$ recognized by $\mathcal{A}$ is the set of all trees accepted by $\mathcal{A}$. A tree language L is *recognizable* if there exists a FTA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = L$.
Now we recall some notions of tree language operations. The tree substitution of the constant c by the language $L \in T_\Sigma$ in the tree $t \in T_\Sigma$, denoted by $t\{c \leftarrow L\}$, is the tree language: i) L if $t = c$; ii) $\{d\}$ if $t = d$ where $d \in \Sigma_0$ and $(d \neq c)$; iii) $f(t_1\{c \leftarrow L\}, \ldots, t_n\{c \leftarrow L\})$ if $t = f(t_1, \ldots, t_n)$. Notice that for $L_1, \ldots, L_n \subseteq T_\Sigma$, $f(L_1, \ldots, L_n)$ is the tree language $\{f(t_1, \ldots, t_n) \mid t_i \in L_i \text{ for } i = 1, \ldots, n\}$.
Then the c−product language $L_1 \cdot_c L_2$ of two languages $L_1, L_2 \in T_\Sigma$ can be defined as:

$$L_1 \cdot_c L_2 = \bigcup_{t\in L_1} \{t\{c \leftarrow L_2\}\}$$

The iterated *c−product* is defined for $L \in T_\Sigma$ as: $L^{0_c} = \{c\}$ and $L^{(n+1)_c} = L^{n_c} \cup L \cdot_c L^{n_c}$. The c−closure of L is defined as $L^{*_c} = \bigcup_{n\geq 0} L^{n_c}$, c is called the symbol of the c−closure operator.

The writing of *regular tree expressions* over Σ respect the following syntax: $\mathrm{E} = f(\underbrace{\mathrm{E}, \mathrm{E}, \cdots, \mathrm{E}}_{n\ times}) \mid \mathrm{E} + \mathrm{E} \mid \mathrm{E} \cdot_c \mathrm{E} \mid \mathrm{E}^{*_c} \mid c,$ with $c \in \Sigma_0 \wedge f \in \Sigma_{\geq 1}$.
The alphabetic width $\|\mathrm{E}\|$ of E is the number of occurrences of symbols of Σ in E. *The size* of E, $|\mathrm{E}|$ is the size of its syntax tree T_E.
The language $[\![\mathrm{E}]\!]$ denoted by the regular expression E is defined inductively as:

$$[\![c]\!] = \{c\} \text{ for all } c \in \Sigma_0,\ [\![f(\mathrm{E}_1, \mathrm{E}_2, \cdots, \mathrm{E}_n)]\!] = f([\![\mathrm{E}_1]\!], \ldots, [\![\mathrm{E}_n]\!]),$$

$$[\![\mathrm{F} + G]\!] = [\![\mathrm{F}]\!] \cup [\![G]\!],\ [\![\mathrm{F} \cdot_c G]\!] = [\![\mathrm{F}]\!] \cdot_c [\![G]\!],\ [\![\mathrm{F}^{*_c}]\!] = [\![\mathrm{F}]\!]^{*_c}$$

It is well known that a tree language is recognizable if and only if it can be denoted by a tree regular expression [7,4].

3 Linearized Regular Tree Expression

In order to define the notion of linearized regular expression, we first introduce two new regular operators: c−product list $\cdot_{<c_1,\ldots,c_n>}$ and c−closure list $*_{<c_1,\ldots,c_n>}$, for tree languages $L, L_1, L_2 \subseteq T_\Sigma$ as follows:

$$L_1 \cdot_{<c_1>} L_2 = L_1\{c_1 \leftarrow L_2\},\ L_1 \cdot_{<c_1,\cdots,c_n>} L_2 = (L_1 \cdot_{<c_1,\cdots,c_{n-1}>} L_2) \cdot_{<c_n>} L_2$$

$$L^{*<c_1\cdots c_n>} : L^{0<c_1\cdots c_n>} = \{c_1\},\ L^{n+1<c_1\cdots c_n>} = L^{n<c_1,\cdots,c_n>}$$

$$\bigcup L \cdot_{<c_1,\cdots,c_n>} L^{n<c_1,\cdots,c_n>},\ L^{*<c_1\cdots c_n>} = \bigcup_{i\geq 0} L^{i<c_1\cdots c_n>}$$

The marked form $\mathrm{E}^{'}$ of a regular expression E is obtained from E by marking differently all symbols (letters and closure symbols). The set of marked symbols are called *positions*. A mapping h is defined from $\mathrm{Pos}_\mathrm{E}(\mathrm{E})$ to Σ. It associates to a marked symbol $a_j \in \mathrm{Pos}_\mathrm{E}(\mathrm{E})$ the symbol $a = h(a_j) \in \Sigma$. We extend h from $T_{\mathrm{Pos}_\mathrm{E}(\mathrm{E})}$ to T_Σ by replacing every position a_j by $h(a_j)$ in trees of $T_{\mathrm{Pos}_\mathrm{E}(\mathrm{E})}$.

For a subexpression F of E, we denote by $\mathrm{Pos}_\mathrm{E}(\mathrm{F})$ the set of positions of E appearing in F. For example if $\mathrm{E} = f(c, g(c)^{*_c})^{*_c}$ then, $\mathrm{E}^{'} = f_1(c_2, g_3(c_4)^{*_{c_5}})^{*_{c_6}}$ and $\mathrm{Pos}_\mathrm{E}(\mathrm{E}) = \{f_1, c_2, g_3, c_4, c_5, c_6\}$. Let F be a subexpression of E and $\mathrm{F}^{'}$ its marked form in E. We denote by $c_\mathrm{F} := < c_{i_1}, \ldots, c_{i_n} >$ with $\{c_{i_1}, \ldots, c_{i_n}\}$ is the set of positions of F in E such that $h(c_{i_j}) = c$ and $c_{i_1} = c_k$ when $\mathrm{F}^{'} = H^{*_{c_k}}$. The linearized regular expression $\overline{\mathrm{E}}$ of E is obtained from $\mathrm{E}^{'}$ by replacing each subexpression $(\mathrm{F}^{'} \cdot_c G^{'})$ (resp. $H^{'*_c}$) in $\mathrm{E}^{'}$ by $\underline{\ }(\mathrm{F}^{'} \cdot_{c_\mathrm{F}} G^{'})$ (resp. $H^{'*_{c_{H^*}}}$. By $\overline{\mathrm{F}}$ we denote the subexpression associated to F in $\overline{\mathrm{E}}$.

In the following, we will denote by $\overline{a}$ the position associated to the symbol a in E.

The semantic $[\![\overline{\mathrm{E}}]\!]$ of a regular expression $\overline{\mathrm{E}}$ is defined inductively by:

$$\mathrm{E} = c,\ [\![\overline{c}]\!] = \{\overline{c}\},\ \mathrm{E} = \mathrm{F}^{*_c},\ [\![\overline{\mathrm{F}^{*_c}}]\!] = [\![\overline{\mathrm{F}}]\!]^{*_{c_{\mathrm{F}^*}}}$$

$$\mathrm{E} = f(\mathrm{E}_1, \dots, \mathrm{E}_n),\ [\![f_1(\overline{\mathrm{E}_1}, \overline{\mathrm{E}_2}, \dots, \overline{\mathrm{E}_n})]\!] = f_1([\![\overline{\mathrm{E}_1}]\!], \dots, [\![\overline{\mathrm{E}_n}]\!])$$

$$\mathrm{E} = \mathrm{F} + G,\ [\![\overline{\mathrm{F} + G}]\!] = [\![\overline{\mathrm{F}}]\!] \cup [\![\overline{G}]\!],\ \mathrm{E} = \mathrm{F} \cdot_c G,\ [\![\overline{\mathrm{F} \cdot_c G}]\!] = [\![\overline{\mathrm{F}}]\!] \cdot_{c_\mathrm{F}} [\![\overline{G}]\!]$$

Example 1. Let $\mathrm{E} = ((c + a) + (g(c))^{*_c})^{*_c} \cdot_c f(a, h(c))$. Then let $\mathrm{E} = \mathrm{F} \cdot_c G$ such that: $\mathrm{F} = ((c + a) + (g(c))^{*_c})^{*_c}$, $G = f(a, h(c))$ and $\mathrm{H} = g(c)$. We have: $\mathrm{E}^{'} = ((c_1 + a_2) + (g_3(c_4))^{*_{c_5}})^{*_{c_6}} \cdot_c f_7(a_8, h_9(c_{10}))$, $\mathrm{Pos}_\mathrm{E}(\mathrm{E}) = \{c_1, a_2, g_3, c_4, c_5, c_6, f_7, a_8, h_9, c_{10}\}$, $c_{\mathrm{F}^*} =< c_6, c_5, c_4, c_1 >$ and $c_{H^*} =< c_5, c_4 >$. The linearized form of E is $\overline{\mathrm{E}} = ((c_1 + a_2) + (g_3(c_4))^{*_{c_{H^*}}})^{*_{c_{\mathrm{F}^*}}} \cdot_{c_{\mathrm{F}^*}} f_7(a_8, h_9(c_{10}))$, $\overline{G} = f_1(a_2, h_3(c_4))$ $\mathrm{Pos}_\mathrm{G}(\mathrm{G}) = \{f_1, a_2, h_3, c_4\}$, $\overline{G}^E = f_7(a_8, h_9(c_{10}))$, $\mathrm{Pos}_\mathrm{E}(\mathrm{G}) = \{f_7, a_8, h_9, c_{10}\}$. The language denoted by $\overline{\mathrm{E}}$ is: $[\![\overline{\mathrm{E}}^E]\!] = \{\ a_2,\ f_7(a_8, h_9(c_{10})), g_3(f_7(a_8, h_9(c_{10}))), g_3(g_3(f_7(a_8, h_9(c_{10})))),\ \ g_3(g_3(g_3(f_7(a_8, h_9(c_{10}))))), \dots\}$.

For a subexpression F of E we will denote by $\overline{\mathrm{F}}$ the subexpression $\overline{\mathrm{F}}^\mathrm{E}$. As has already been the case for words, the linearized expression $\overline{\mathrm{E}}$ of E should verify $h([\![\overline{\mathrm{E}}]\!]) = [\![\mathrm{E}]\!]$.

Proposition 2. *Let* E *be a regular expression and* $\overline{\mathrm{E}}$ *its linearized form. Then we have:*

$$h([\![\overline{\mathrm{E}}]\!]) = [\![\mathrm{E}]\!]$$

Proof. The proof of this Proposition is done by induction on the structure of E. If $\mathrm{E} = a$ with $a \in \Sigma_0$ and $\overline{a} \in \mathrm{Pos}_\mathrm{E}(\mathrm{E})$, we have $h(\overline{a}) = a$ and $a \in [\![\mathrm{E}]\!]$. Supposing now that the property is true for subexpressions $\mathrm{F}, G, \mathrm{E}_1, \dots, \mathrm{E}_m$. It should be proved that the property is true for the expressions $\mathrm{F} + G$, $\mathrm{F} \cdot_c G$, F^{*_c} and $f(\mathrm{E}_1, \cdots, \mathrm{E}_m)$. We have to prove that $h([\![\overline{\mathrm{E}}]\!]) \subseteq [\![\mathrm{E}]\!]$ and $[\![\mathrm{E}]\!] \subseteq h([\![\overline{\mathrm{E}}]\!])$.
Here we only give the proof for the cases $\mathrm{F} + G$ and $\mathrm{F} \cdot_c G$.
<u>Case $\mathrm{E} = \mathrm{F} + G$:</u> Let $t \in [\![\mathrm{F} + G]\!]$. By the induction hypothesis, we have $[\![\mathrm{F}]\!] = h([\![\overline{\mathrm{F}}]\!])$ and $[\![G]\!] = h([\![\overline{G}]\!])$. This means that $t \in h([\![\overline{\mathrm{F}}]\!]) \cup h([\![\overline{G}]\!])$. This also means that a term $\overline{t}$ such that $\overline{t} \in [\![\overline{\mathrm{F}}]\!]$ or $\overline{t} \in [\![\overline{G}]\!]$ with $h(\overline{t}) = t$ exists. Then $\overline{t} \in [\![\overline{\mathrm{F}}]\!] \cup [\![\overline{G}]\!] = [\![\overline{\mathrm{F} \cup G}]\!]$. Therefore $t \in h([\![\overline{\mathrm{F} + G}]\!])$.
Let $t \in h([\![\overline{\mathrm{F} + G}]\!])$. Thus $t \in h([\![\overline{\mathrm{F}}]\!]) \cup h([\![\overline{G}]\!])$. Which means that $t \in h([\![\overline{\mathrm{F}}]\!])$ or $t \in h([\![\overline{G}]\!])$. By induction hypothesis we have $t \in [\![\mathrm{F}]\!]$ or $t \in [\![G]\!]$. Therefore $t \in [\![\mathrm{F} + G]\!]$.
<u>Case $\mathrm{E} = \mathrm{F} \cdot_c G$:</u> Let $t \in [\![\mathrm{F}]\!] \cdot_c [\![G]\!]$. In this case t is in a set $t_f \cdot_c \{t_{g_1}, \dots, t_{g_k}\}$ where $t_f \in [\![\mathrm{F}]\!]$, and $t_{g_1}, \dots, t_{g_k} \in [\![G]\!]$. Therefore $t_f \in h([\![\overline{\mathrm{F}}]\!])$ and $t_{g_i} \in h([\![\overline{G}]\!]), 1 \leq i \leq k$. Trees $\overline{t_f} \in [\![\overline{\mathrm{F}}]\!]$ and $\overline{t_{g_i}} \in [\![\overline{G}]\!]$, $1 \leq i \leq k$ exist with $h(\overline{t_f}) = t_f$ and $h(\overline{t_{g_i}}) = t_g, i = 1 \dots k$.
Then $c_\mathrm{F} =< c_{i_1}, \dots, c_{i_k} >$ exists with $c_{i_j} \in \mathrm{Pos}_\mathrm{E}(\mathrm{F}), 1 \leq j \leq k$ with $h(c_{i_j}) = c$, such that $\overline{t} \in (((\,[\![\overline{\mathrm{F}}]\!]) \cdot_{c_{i_1}} ([\![\overline{G}]\!])) \cdot_{c_{i_2}} (([\![\overline{G}]\!]) \dots) \cdot_{i_k} [\![\overline{G}]\!])$. We have $\overline{t} \in [\![\overline{\mathrm{F}}]\!] \cdot_{c_\mathrm{F}} [\![\overline{G}]\!] = [\![\overline{\mathrm{F} \cdot_c G}]\!]$. By induction hypothesis, $\overline{t} \in ([\![\overline{\mathrm{F} \cdot_c G}]\!])$. We also prove that for every $t \in h([\![\overline{\mathrm{F} \cdot_c G}]\!])$ it implies $t \in [\![\mathrm{F} \cdot_c G]\!]$

4 Position Tree Automaton

The set of positions associated to E are straightforwardly deduced from the set of letters associated to E. In order to construct a non−deterministic finite automaton associated to the tree expression E that recognizes $[\![\mathrm{E}]\!]$, we introduce the following position sets: $\mathrm{First}(\mathrm{E})$, $\mathrm{Last}(\mathrm{E},\overline{f})$ and $\mathrm{Follow}(\mathrm{E},\overline{f})$.

Definition 3. *Let* E *be a regular tree expression. The operator* $\lambda_c(\mathrm{E})$*, with* $c \in \Sigma_0$*, is defined as follows:*

$$\lambda_c(\mathrm{E}) = \begin{cases} 1 & \textit{if } c \in [\![\mathrm{E}]\!] \\ 0 & \textit{otherwise} \end{cases}$$

The function $\lambda_c(\mathrm{E})$ is a boolean, it values 0 or 1, depending on whether the constant c belongs to the language $[\![\mathrm{E}]\!]$ or not. In what follows, we propose a method of computing $\lambda_c(\mathrm{E})$.

Proposition 4. *The computation of* $\lambda_c(\mathrm{E})$ *is done inductively as follows:*

$$\begin{aligned}
\lambda_c(a) &= \begin{cases} 1 & \textit{if } a = c \\ 0 & \textit{otherwise} \end{cases} \\
\lambda_c(\mathrm{F}+G) &= \lambda_c(\mathrm{F}) \vee \lambda_c(G) \\
\lambda_c(\mathrm{F}\cdot_x G) &= \begin{cases} \lambda_c(\mathrm{F}) \wedge \lambda_c(G) & \textit{if } x = c \\ \lambda_c(\mathrm{F}) \vee (\lambda_x(\mathrm{F}) \wedge \lambda_c(G)) & \textit{otherwise} \end{cases} \\
\lambda_c(\mathrm{F}^{*_x}) &= \begin{cases} 1 & \textit{if } x = c \\ \lambda_c(\mathrm{F}) & \textit{otherwise} \end{cases} \\
\lambda_c(g(\mathrm{E}_1,\cdots,\mathrm{E}_m)) &= 0
\end{aligned}$$

In the following we denote by $\overline{f}$ the position associated to the letter f in E. We define $\mathrm{Last}(\mathrm{E},\overline{f})$ as the set of children of subtrees rooted at f in $[\![\mathrm{E}]\!]$.

Proposition 5. *The function* $\mathrm{Last}(\mathrm{E},\overline{f})$ *is computed inductively as follows:*

$$\begin{aligned}
\mathrm{Last}(a,\overline{f}) &= \emptyset \\
\mathrm{Last}(\mathrm{F}+G,\overline{f}) &= \mathrm{Last}(\mathrm{F},\overline{f}) \uplus \mathrm{Last}(G,\overline{f}) \\
\mathrm{Last}(\mathrm{F}\cdot_x G,\overline{f}) &= \begin{cases} \mathrm{Last}(\mathrm{F},\overline{f}) \cdot_x (h(\mathrm{First}(G)) \cap \Sigma_0) \ \textit{if } \overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{F}) \\ \mathrm{Last}(G,\overline{f}) \ \textit{if } f \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{G}) \end{cases} \\
\mathrm{Last}(\mathrm{F}^{*_x},\overline{f}) &= \mathrm{Last}(\mathrm{F},\overline{f}) \cdot_x (h(\mathrm{First}(\mathrm{F}^{*_x})) \cap \Sigma_0) \\
\mathrm{Last}(g(\mathrm{E}_1,\cdots,\mathrm{E}_m),\overline{f}) &= \begin{cases} \bigcup_{i=1\ldots m}(h(\mathrm{First}(\mathrm{E}_i)) \cap \Sigma_0) \ \textit{if } g = h(\overline{f}) \\ \mathrm{Last}(\mathrm{E}_k,\overline{f}) \ \textit{if } \overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{E_k}) \end{cases}
\end{aligned}$$

For example $\mathrm{Last}(f(a,g(c),b)^{*_c}, f_1) = \{a,b\}$.

To define the position tree automaton we need to define two sets, the set $\mathrm{First}(\mathrm{E})$ and the set $\mathrm{Follow}(\mathrm{E},\overline{f})$.

Definition 6. *The set* $\mathrm{First}(\mathrm{E})$ *contains the positions of the expression* E *which begin at less one term of* $[\![\overline{\mathrm{E}}]\!]$.

The set First(E) of positions is described in the following way. A symbol $\overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{E})$ belongs to the set First(E) if there exists $t_1, \ldots, t_m$ $(m \geq 0)$ such that $\overline{f}(t_1, \cdots t_m) \in [\![\overline{\mathrm{E}}]\!]$.

Let l be a set of tuples $(f_1, \cdots, f_m) \in (\mathrm{Pos}_{\mathrm{E}}(\mathrm{E}))^m$ and $l^{'}$ be a set of positions in $\mathrm{Pos}_{\mathrm{E}}(\mathrm{E})$. The operation $l \cdot_{c_{\mathrm{F}}} l^{'}$ is the set obtained by all the substitution of c_i in the list c_{F} in each tuple of l by elements of $l^{'}$ such that $h(c_i) = c$.

Definition 7. *The set* $\mathrm{Follow}(\mathrm{E}, \overline{f})$ *is composed of* m *tuples* $(f_1, \ldots, f_m) \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{E})^m$ *such that there exists* $t_1, \cdots, t_m \in T_{\mathrm{Pos}_{\mathrm{E}}(\mathrm{E})}$ *whose root of the term* $t_i (1 \leq i \leq m)$ *is position* f_i *and* $\overline{f}(t_1, \cdots, t_m)$ *is a subtree of a tree in* $[\![\overline{\mathrm{E}}]\!]$.

After defining the sets First(E) and $\mathrm{Follow}(\mathrm{E}, \overline{f})$, we now describe how to compute these sets.

Proposition 8. *Let* H *be a subexpression of* E. *The set of positions* First(H) *can be computed inductively as follows:*

$$\begin{aligned}
\mathrm{First}(a) &= \{\overline{a}\} \\
\mathrm{First}(\mathrm{F} + G) &= \mathrm{First}(\mathrm{F}) \cup \mathrm{First}(G) \\
\mathrm{First}(\mathrm{F} \cdot_c G) &= \begin{cases} \mathrm{First}(\mathrm{F}) \cdot_{c_{\mathrm{F}}} \mathrm{First}(G) & \textit{if } \lambda_c(\mathrm{F}) = 1 \\ \mathrm{First}(\mathrm{F}) & \textit{otherwise} \end{cases} \\
\mathrm{First}(\mathrm{F}^{*_c}) &= \mathrm{First}(\mathrm{F}) \cdot_{c_{\mathrm{F}^*}} \{\overline{c_*}\} \cup \{\overline{c_*}\} \\
\mathrm{First}(f(\mathrm{E}_1, \cdots, \mathrm{E}_m)) &= \{\overline{f}\}
\end{aligned}$$

Proposition 9. *The set of tuples* $\mathrm{Follow}(\mathrm{E}, \overline{f})$ *can be computed inductively as follows:*

$$\begin{aligned}
\mathrm{Follow}(\mathrm{E}, \overline{f}) &= \emptyset \\
\mathrm{Follow}(\mathrm{F} + G, \overline{f}) &= \mathrm{Follow}(\mathrm{F}, \overline{f}) \uplus \mathrm{Follow}(G, \overline{f}) \\
\mathrm{Follow}(\mathrm{F} \cdot_c G, \overline{f}) &= \begin{cases} \mathrm{Follow}(\mathrm{F}, \overline{f}) \ \textit{if } \overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{F}) \wedge \\ c \notin \mathrm{Last}(\mathrm{F}, \overline{f}) \\ \mathrm{Follow}(\mathrm{F}, \overline{f}) \cdot_{c_{\mathrm{F}}} \mathrm{First}(G) \ \textit{if} \\ c \in \mathrm{Last}(\mathrm{F}, \overline{f}) \wedge \overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{F}) \\ \mathrm{Follow}(G, \overline{f}) \ \textit{if } \overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{G}). \end{cases} \\
\mathrm{Follow}(\mathrm{F}^{*_c}, \overline{f}) &= \mathrm{Follow}(\mathrm{F}, \overline{f}) \cdot_{c_{\mathrm{F}^*}} \mathrm{First}(\mathrm{F}^{*_c}) \\
\mathrm{Follow}(f(\mathrm{E}_1, \ldots, \mathrm{E}_m), \overline{f}) &= \mathrm{First}(\mathrm{E}_1) \times \cdots \times \mathrm{First}(\mathrm{E}_m)
\end{aligned}$$

Note that the above Proposition 8 and Proposition 9 proof that allow the computation of functions First and Follow were made by induction on the structure of the regular tree expression E.

Now that we have defined the functions First and Follow. First the position tree automaton $\mathcal{P}_{\overline{\mathrm{E}}}$ associated with $\overline{\mathrm{E}}$ is introduced and we show that it recognizes the language $[\![\overline{\mathrm{E}}]\!]$. Next, its image by the mapping h, $\mathcal{P}_{\mathrm{E}}$ associated to the expression E, recognizes the language $[\![\mathrm{E}]\!]$ is shown.

Definition 10. *Let* E *be a regular expression and* $\overline{\mathrm{E}}$ *its linearized form. The automaton* $\mathcal{P}_{\overline{\mathrm{E}}} = (Q, \Sigma, Q_T, \Delta)$ *is defined using* $\mathrm{First}(E)$ *and* $\mathrm{Follow}(\mathrm{E}, \overline{f})$*:*

- $Q = \{q_{\overline{\alpha}} \mid \overline{\alpha} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{E})\}$*: the set of states,*
- $Q_T = \{q_{\overline{\alpha}} \mid \overline{\alpha} \in \mathrm{First}(\mathrm{E})\}$*: the set of final states,*
- $\Delta = \{\ \{(q_{\overline{f}}, \overline{f}, q_{f_1}, \ldots, q_{f_m}) \mid (f_1, \ldots, f_m) \in \mathrm{Follow}(E, \overline{f}),$
  $h(\overline{f}) \in \Sigma_m\} \cup \{(q_{\overline{x}}, \overline{x}) \mid h(\overline{x}) \in \Sigma_0\}\ \}$*: the set of transition rules.*

Theorem 11. *Let* E *be a regular expression and* $\overline{\mathrm{E}}$ *its linearized form. The automaton* $\mathcal{P}_{\overline{\mathrm{E}}}$ *recognizes* $[\![\overline{\mathrm{E}}]\!]$*.*

Proof. The proof of this theorem ($L(\mathcal{P}_{\overline{\mathrm{E}}}) = [\![\overline{E}]\!]$) is done by induction on the structure of E.
Case $\mathrm{E} = a$: we have $\overline{a} \in [\![\overline{E}]\!]$. On the other hand, $\mathrm{First}(\mathrm{E}) = \{\overline{a}\}$, $Q = Q_T = \{q_{\overline{a}}\}$ are the set of states and final states and $\Delta = \{(\overline{a}, q_{\overline{a}})\}$ the set of transition rules. Therefore $\overline{a}$ is recognized by the language associated to the automaton $\mathcal{P}_{\overline{\mathrm{E}}} = (Q, Q_T, \{\overline{a}\}, \Delta)$. The theorem is true for the base case. If we assume that it is true for the expressions F, G i.e we have $L(\mathcal{P}_{\overline{\mathrm{F}}}) = [\![\overline{\mathrm{F}}]\!]$, $L(\mathcal{P}_{\overline{G}}) = [\![\overline{G}]\!]$, and prove it for the expressions $\mathrm{E} = \mathrm{F} + G$ and $\mathrm{E} = \mathrm{F} \cdot_c G$.
Case $\mathrm{E} = \mathrm{F} + G$: Let $t \in [\![\overline{\mathrm{E}}]\!]$. We have by definition, $[\![\overline{\mathrm{E}}]\!] = [\![\overline{\mathrm{F} + G}]\!] = [\![\overline{\mathrm{F}}]\!] \cup [\![\overline{G}]\!]$. Since $\overline{\mathrm{E}}$ is linear, we have $t \in [\![\overline{\mathrm{F}}]\!]$ or $t \in [\![\overline{G}]\!]$.
If $t \in [\![\overline{\mathrm{F}}]\!]$, then by induction hypothesis we have $t \in [\![\overline{\mathrm{F}}]\!]$ if the term t is in $\mathcal{L}(\mathcal{P}_{\overline{\mathrm{F}}})$. Therefore a path exists in the automaton $\mathcal{P}_{\overline{\mathrm{F}}}$ composed of transition rules that recognizes the term t. The set of transition rules of the automaton $\mathcal{P}_{\overline{\mathrm{F}}}$ denoted by $\Delta_{\mathrm{F}} = \{(q_{\overline{f}}, \overline{f}, q_{h_1}, \cdots, q_{h_m}) \mid (h_1, \cdots, h_m) \in \mathrm{Follow}(\mathrm{F}, \overline{f}), \overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{F}) \wedge h(\overline{f}) \in \Sigma_{\geq 1}\} \cup \{(q_x, x) \mid h(x) \in \Sigma_0\}\} \subseteq \{(q_{\overline{f}}, \overline{f}, q_{h_1}, \cdots, q_{h_m}) \mid (h_1, \cdots, h_m) \in \mathrm{Follow}(\mathrm{E}, \overline{f}), \overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{E}) \wedge h(\overline{f}) \in \Sigma_{\geq 1}\} \cup \{(q_x, x) \mid h(x) \in \Sigma_0\}\}$ and a sequence $\mathrm{First}(\mathrm{F}) \subseteq \mathrm{First}(\mathrm{E})$, then there exists a path in $\mathcal{P}_{\overline{\mathrm{E}}}$ labeled by t. Whereas, $t \in \mathcal{L}(\mathcal{P}_{\overline{\mathrm{E}}})$.
If $t \in [\![\overline{G}]\!]$, $t \in [\![\overline{G}]\!]$ if and only if $t \in L(\mathcal{P}_{\overline{G}})$. The term t is recognized by the automaton $\mathcal{P}_{\overline{G}}$. Then a path exists in $\mathcal{P}_{\overline{\mathrm{F}}}$ that recognizes t. We have $\Delta_G = \{(q_{\overline{f}}, \overline{f}, q_{h1}, \cdots, q_{hm}) \mid (h_1, \cdots, h_m) \in \mathrm{Follow}(G, \overline{f}), \overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{G}) \wedge h(\overline{f}) \in \Sigma_{\geq 1}\}$ $\cup \{(q_x, x) \mid h(x) \in \Sigma_0\}\} \subseteq \{(q_{\overline{f}}, \overline{f}, q_{h1}, \cdots, q_{hm}) \mid (h_1, \cdots, h_m) \in \mathrm{Follow}(\mathrm{E}, \overline{f}),$ $\overline{f} \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{E}) \wedge h(f) \in \Sigma_{\geq 1}\} \cup \{(q_x, x) \mid h(x) \in \Sigma_0\}\}$. We have $\mathrm{First}(G) \subseteq \mathrm{First}(\mathrm{E})$. Then a path exists in $\mathcal{P}_{\overline{\mathrm{E}}}$ labeled by t. Therefore $t \in \mathcal{L}(\mathcal{P}_{\overline{\mathrm{E}}})$.
Case $\mathrm{E} = \mathrm{F} \cdot_c G$ and $G \neq c$: We have $[\![\overline{\mathrm{E}}]\!] = [\![\overline{\mathrm{F} \cdot_c G}]\!] = [\![\overline{\mathrm{F}}]\!] \cdot_{<c_1, \cdots, c_n>} [\![\overline{G}]\!]$. Let $t \in [\![\overline{\mathrm{F} \cdot_c G}]\!]$ so $t \in [\![\overline{\mathrm{F}}]\!] \cdot_{<c_1, \cdots, c_n>} [\![\overline{G}]\!]$. Then a term $t_{\mathrm{F}} \in [\![\overline{\mathrm{F}}]\!]$ exists and a set of terms $t_G^1, \ldots t_G^m \in [\![\overline{G}]\!]$ such that the term $t \in \{t_{\mathrm{F}}\} \cdot_{<c_1, \cdots, c_n>} \{t_G^1, \ldots t_G^m\}$. Three cases are possible.
Case 1: The leaves of t_{F} are all different from the positions c_i for $i = 1, \ldots, m$. In this case, the term t belongs necessarily to the language $[\![\overline{\mathrm{F}}]\!]$. By induction hypothesis, we have $t \in \mathcal{L}(\mathcal{P}_{\overline{\mathrm{F}}})$. For any transition rule of the form (q_x, x) such that $x \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{F})\}$ in the automaton $\mathcal{P}_{\overline{\mathrm{F}}}$, (q_x, x) is a transition rule in the automaton $\mathcal{P}_{\overline{\mathrm{E}}}$. Indeed, $\{(q_x, x) | x \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{F})\} \subseteq \{(q_x, x) | x \in \mathrm{Pos}_{\mathrm{E}}(\mathrm{E})\}$. Let $(q_f, \overline{f}, q_{h_1}, \ldots, q_{h_n})$ a transition rule in the automaton $\mathcal{P}_{\overline{\mathrm{F}}}$ that recognizes the term t. Then, there exists a path in $\mathcal{P}_{\overline{\mathrm{F}}}$ labeled by t. This means that

$(h_1, \ldots, h_m) \in \text{Follow}(\text{F}, \overline{f})$. As $\text{Follow}(\text{E}, \overline{f}) = \text{Follow}(\text{F}, \overline{f}) \cdot_{<c_1, \cdots, c_n>} \text{First}(G)$ $\uplus \text{Follow}(G, \overline{f})$, we have $\text{Follow}(\text{E}, \overline{f}) = \text{Follow}(\text{F}, \overline{f})$ (because we have $t \in [\![\overline{\text{F}}]\!]$) and $(h_1, \ldots, h_m) \in \text{Follow}(\text{E}, \overline{f})$ from the fact that $h_i \neq c_j$ for $1 \leq i \leq m$ and $1 \leq j \leq k$. Therefore $(q_f, \overline{f}, q_{h_1}, \ldots, q_{h_n})$ is a transition rule in the automaton $\mathcal{P}_{\overline{\text{E}}}$. Then we have proved that $t \in \mathcal{L}(\mathcal{P}_{\overline{\text{E}}})$.
Case 2: the term $t \in [\![\overline{G}]\!]$ this means that $t_\text{F} = c_i$. It is obvious to show that $t \in [\![\overline{\text{E}}]\!]$. By induction hypothesis we have $t \in \mathcal{L}(\mathcal{P}_{\overline{\text{E}}})$.
Case 3: The term $t \in \{t_\text{F}\} \cdot_{<c_1, \cdots, c_n>} \{t_G^1, \ldots t_G^m\}$. The root of the term t_F is of arity strictly greater than 0 and t_F contains at least one position $c_i \in \text{Pos}_\text{E}(\text{F})$. Let $(q_f, f, q_{h_1}, \cdots, q_{h_m})$ be a transition rule in the path recognizing the term t_F in the automaton $\mathcal{P}_{\overline{\text{F}}}$. This means that there exist a tuple $(h_1, \ldots, c_i, \ldots, h_m) \in \text{Follow}(\text{F}, \overline{f})$. The term t is constructed from t_F by substituting each c_i by an element of the set $\{t_G^1, \ldots t_G^m\}$. Thus, $(h_1, \ldots, \mathit{root}\,(t_G^{c_i}), \ldots, \mathit{root}\left(t_G^{c_j}\right), \ldots h_m) \in \text{Follow}(\text{E}, \overline{f})$ with $\mathit{root}\,(t_G^{c_i})$ is the root of the term obtained by substituting c_i by one term of $\{t_G^1, \ldots t_G^m\}$. Then we show that there exist a transition rule $(q_l, l, q_{h_1}, \ldots, q_{\mathit{root}(t_G^{c_i})}, \ldots, q_{\mathit{root}\left(t_G^{c_j}\right)}, \ldots q_{h_m})$ in the automaton $\mathcal{P}_{\overline{\text{E}}}$. Therefore, the term $t \in \mathcal{L}(\mathcal{P}_{\overline{\text{E}}})$.

One can prove in the similar way that if t is accepted by $\mathcal{P}_{\overline{\text{E}}}$ then t is in $[\![\overline{\text{E}}]\!]$.

The automaton $\mathcal{P}_\text{E}$ is obtained from the automaton $\mathcal{P}_{\overline{\text{E}}}$ by remplacing each position $\overline{x}$ by the letter $h(\overline{x}) = x$. Positions of the expression E are the states of the position tree automaton of $\mathcal{P}_\text{E}$.

Definition 12. *Let* E *be a regular tree expression. The position tree automaton* $\mathcal{P}_\text{E} = (Q, \Sigma, Q_T, \Delta)$ *associated to the tree expression* E *is defined as follows:*

- $Q = \{q_{\overline{\alpha}} \mid \overline{\alpha} \in \text{Pos}_\text{E}(\text{E})\}$*: the set of states,*
- $Q_T = \{q_{\overline{\alpha}} \mid \overline{\alpha} \in \text{First}(\text{E})\}$*: the set of final states,*
- $\Delta = \{\{(q_{\overline{f}}, h(\overline{f}), q_{f_1}, \ldots, q_{f_m}) \mid (f_1, \ldots, f_m) \in \text{Follow}(\text{E}, \overline{f}),$ $h(\overline{f}) \in \Sigma_m\} \cup \{(q_{\overline{x}}, h(\overline{x})) \mid h(\overline{x}) \in \Sigma_0\}\}$*: the set of transition rules.*

Example 13. Let $\text{E} = ((c + a) + (g(c))^{*_c})^{*_c} \cdot_c f(a, h(c))$ the expression of the example 1. We have

$\text{First}\,(\text{E}) = \{a_2, g_3, f_7\}$
$\text{Follow}\,(\text{E}, g_3) = \{(g_3), (f_7)\}$ $\text{Follow}\,(\text{E}, h_9) = \{(c_{10})\}$ $\text{Follow}\,(\text{E}, f_7) = \{(a_8, h_9)\}$

The set of states is $Q = \{q_{c_1}, q_{a_2}, q_{g_3}, q_{c_4}, q_{c_5}, q_{c_6}, q_{f_7}, q_{a_8}, q_{h_9}, q_{c_{10}}\}$.
The set of final states is $Q_T = \{q_{a_2}, q_{g_3}, q_{f_7}\}$
The set of transition rules consists of

$$\begin{array}{llll}
c \rightarrow q_{c_1} & a \rightarrow q_{a_2} & c \rightarrow q_{c_4} & c \rightarrow q_{c_5} \\
c \rightarrow q_{c_6} & a \rightarrow q_{a_8} & c \rightarrow q_{c_{10}} & h(q_{c_{10}}) \rightarrow q_{h_9} \\
g(q_{g_3}) \rightarrow q_{g_3} & g(q_{f_7}) \rightarrow q_{g_3} & f(q_{a_8}, q_{h_9}) \rightarrow q_{f_7} &
\end{array}$$

Corollary 14. *Let* E *be a regular tree expression,* $\overline{\text{E}}$ *its linearized form and* $\mathcal{P}_{\overline{\text{E}}}$ *the automaton associated to* $\overline{\text{E}}$*, then:*

$$\mathcal{L}(h(\mathcal{P}_{\overline{\text{E}}})) = h([\![\overline{\text{E}}]\!])$$

The construction of the position tree automaton $\mathcal{P}_{\mathrm{E}}$ from a regular tree expression as it has been defined in this article complies with the properties of the position automaton proposed by Glushkov. It is homogeneous (for all state $q \in Q$, all transitions entering in q are labeled by the same symbol). This is the generalization of the position automaton from words to trees. The number of transition rules of this automaton is exponential. This is due to the Formula 9 in Proposition 9. In order to reduce the number of these transition rules, we propose an equivalent automaton $\mathcal{R}_{\mathrm{E}}$ of the position tree automaton $\mathcal{P}_{\mathrm{E}}$ associated to E. The idea is to modify the set of transition rules associated to the subexpressions of the form $f(\mathrm{E}_1, \ldots, \mathrm{E}_m)$ and replace computation of the set $\mathrm{Follow}(\mathrm{E}, \overline{f}) = \mathrm{First}(\mathrm{E}_1) \times \cdots \times \mathrm{First}(\mathrm{E}_m)$ by $\mathrm{Follow}(\mathrm{E}, \overline{f}) = \{(\mathrm{First}(\mathrm{E}_1), \ldots, \mathrm{First}(\mathrm{E}_m))\}$. Thus, the following equivalent reduced automaton $\mathcal{R}_{\mathrm{E}}$ is proposed.

Definition 15. *Let* E *be a regular tree expression. We can compute the reduced automaton* $\mathcal{R}_{\mathrm{E}} = (Q, \Sigma, Q_T, \Delta)$ *associated to the expression* E *as follows:*

- $Q = \{q_{\{\alpha\}} \mid \alpha \in \mathrm{First}(\mathrm{E})\} \cup \{q_{\mathrm{First}(\mathrm{E}_k)} \mid g(\mathrm{E}_1, \ldots, \mathrm{E}_n),$ *is the subexpression of* E *and* $1 \leq k \leq n\}$*:the set of states,*
- $Q_T = \{q_{\{\alpha\}} \mid \alpha \in \mathrm{First}(\mathrm{E})\}$*: the set of final states,*
- $\Delta = \{(q_I, h(\overline{f}), q_{J_1}, \ldots, q_{J_l}) \mid (J_1, \ldots, J_l) \in \mathrm{Follow}(\mathrm{E}, \overline{f})$ *and* $\overline{f} \in I$, $h(\overline{f}) \in \Sigma_{\geq 1}\} \cup \{(q_I, h(\overline{x})) \mid \overline{x} \in I$ *and* $h(\overline{x}) \in \Sigma_0\}$*: the set of transition rules.*

It is proved that $\mathcal{L}(\mathcal{R}_{\mathrm{E}}) = [\![\mathrm{E}]\!]$ and that the number of transition rules $|\Delta|$ is less than or equal to $|\,\mathrm{E}\,|^2$.

Example 16. Let $\mathrm{E} = f(a + b, c + d)$. We have: $\overline{\mathrm{E}} = f_1(a_2 + b_3, c_4 + d_5)$, $\mathrm{First}(\mathrm{E}) = \{f_1\}$, $\mathrm{Follow}(\mathrm{E}, f_1) = \{(a_2, c_4), (a_2, d_5), (b_3, c_4), (b_3, d_5)\}$.
The automaton $\mathcal{P}_{\mathrm{E}}$ associated to E is defined as follows:
The set of states is

$$Q_{\mathcal{P}} = \{q_{f_1}, q_{a_2}, q_{b_3}, q_{c_4}, q_{d_5}\}$$

The set of final states is

$$Q_{T_{\mathcal{P}}} = \{q_{f_1}\}$$

The set of transition rules is

$$\begin{array}{ll} a \to q_{a_2} & b \to q_{b_3} \\ c \to q_{c_4} & d \to q_{d_5} \\ f(q_{a_2}, q_{c_4}) \to q_{f_1} & f(q_{a_2}, q_{d_5}) \to q_{f_1} \\ f(q_{b_3}, q_{c_4}) \to q_{f_1} & f(q_{b_3}, q_{d_5}) \to q_{f_1}\} \end{array}$$

We have the size $|Q_{\mathcal{P}}| = 5$ and the number of transion is $|\Delta_{\mathcal{P}}| = 8$.

The modified $\mathrm{Follow}(\mathrm{E}, f_1)$ is equal to $\{(\{a_2, b_3\}, \{c_4, d_5\})\}$.
The reduced automaton $\mathcal{R}_{\mathrm{E}}$ associated to E is defined as follows:
The set of states is

$$Q_{\mathcal{R}} = \{q_{\{f_1\}}, q_{\{a_2, b_3\}}, q_{\{c_4, d_5\}}\}$$

The set of final states is

$$Q_{T_{\mathcal{R}}} = \{q_{\{f_1\}}\}$$

The set of rules consists of

$$\begin{array}{ll} a \to q_{\{a_2,b_3\}} & b \to q_{\{a_2,b_3\}} \\ c \to q_{\{c_4,d_5\}} & d \to q_{\{c_4,d_5\}} \\ f(q_{\{a_2,b_3\}}, q_{\{c_4,d_5\}}) \to q_{\{f_1\}} & \end{array}$$

The number of states $|Q_{\mathcal{R}}| = 3$ and the number of transition rules is $|\Delta_{\mathcal{R}}| = 5$.

5 Algorithm and Complexity

An implicit construction of the position automaton, the so-called ZPC-structure, has been developed by Ziadi et al. [10]. Algorithm 1 extends this construction to the regular tree expressions. It constructs a forest of trees where every tree rooted at a node ν_{F} represents the set First(F).

Algorithm 1. ZPC structure construction

```
for each subexpression $\mathrm{F}^{*_c}$ of E do
replace it by $(\mathrm{F}+c)^{*_c}$;
end for
Construct the syntax tree $T_{\mathrm{E}}$ of E;
for each constant $a \in \Sigma_0$ do
    for each node $\nu_{\mathrm{F}}$ on $T_{\mathrm{E}}$ do
        Compute the values $\lambda_a(\mathrm{F})$
    end for
end for
# The construct of First Forest
for each node $\nu_{\mathrm{F}\cdot_c G}$ in $T_{\mathrm{E}}$ do
    if $\lambda_c(\mathrm{F}) = 0$ then
        Remove the link $(\nu_{\mathrm{F}\cdot_c G}, \nu_G)$
        Set $c$ as label of the edge $(\nu_{\mathrm{F}\cdot_c G}, \nu_{\mathrm{F}})$
    end if
end for
for each node labeled $f \in \Sigma_{\geq 1}$ do
    Compute Last(E, $\overline{f}$)
end for
for each node $\nu_{\mathrm{F}\cdot_c G}$ in $T_{\mathrm{E}}$ do create a follow link from $\nu_{\mathrm{F}}$ to $\nu_G$
end for
for each node $\nu_{\mathrm{F}^{*_c}}$ in $T_{\mathrm{E}}$ do create a link from $\nu_{\mathrm{F}}$ to $\nu_{\mathrm{F}^{*_c}}$
end for
```

The ZPC-structure constructed by Algorithm 1 allows us to efficiently compute the sets Follow(E, $\overline{f}$) for $f \in \Sigma_{\geq 1}$.

Algorithm 2. Transition rules Computation

```
for each $f \in \Sigma_{\geq 1}$ such that $f(\mathrm{E}_1, \ldots, \mathrm{E}_n)$ is a subexpression of E do
    Let $\Gamma(f) = \{(a_1, \nu_{R_1}), (a_2, \nu_{R_2}), \ldots (a_m, \nu_{R_m})\}$
    for $i = 1$ to $n$ do
        $X_i := ((((\mathrm{First}(\mathrm{E}_i) \cdot_{a_1} e(\nu_{R_1})) \cdot_{a_2} e(\nu_{R_2})) \cdot_{a_3} \cdots) \cdot_{a_m} e(\nu_{R_m}))$
    end for
    $\mathrm{Follow}(\mathrm{E}, \overline{f}) := \{(X_1, \ldots, X_n)\}$
end for
```

First(E_i) is the set of positions beginning terms of $[\![\overline{\mathrm{E}_i}]\!]$, such that $\overline{\mathrm{E}_i}$ is marked in E.
$\Gamma(f)$ is the set of Follow Links induced by a c-product $\cdot_c$ or c-closure $*_c$ operator, in the path from the node ν_f associated with the symbol f to the root of $T_{\overline{\mathrm{E}}}$.
The set $e(\nu_{R_i})$ is the First set of some subexpression of E. For a subexpression $\mathrm{F} \cdot_c G$, $e(\nu_{\mathrm{F} \cdot_c G})$ is defined as $e(\nu_{\mathrm{F} \cdot_c G}) := e(\nu_{\mathrm{F}}) \backslash \{c_i \mid h(c_i) = c\} \cup e(\nu_G)$.

The time and space complexities of the computation of the sets Last$(\mathrm{E}, \overline{f})$, First(E) and Follow$(\mathrm{E}, \overline{f})$ are the following.
The formula $X_i := ((((\mathrm{First}(\mathrm{E}_i) \cdot_{a_1} e(\nu_{R_1})) \cdot_{a_2} e(\nu_{R_2})) \cdot_{a_3} \cdots) \cdot_{a_m} e(\nu_{R_m}))$ can be seen as the union. The complexity of computing of the set Follow$(\mathrm{E}, \overline{f})$ is equal to the complexity of the computation of the formula
$X_i := ((((\mathrm{First}(\mathrm{E}_i) \cdot_{a_1} e(\nu_{R_1})) \cdot_{a_2} e(\nu_{R_2})) \cdot_{a_3} \cdots) \cdot_{a_m} e(\nu_{R_m}))$. This union is not necessarily disjointed and can be done in time $O(|\,\mathrm{E}\,| \cdot ||\,\mathrm{E}\,||)$. Therefore, the time complexity of the computation of Follow$(\mathrm{E}, \overline{f})$ for all symbols $f \in \Sigma_{\geq 1}$ is $O(|\Sigma_0| \cdot |\,\mathrm{E}\,| \cdot ||\,\mathrm{E}\,||)$. However, by using the algorithm for deleting redundant links [8], the formula

$$X_i := ((((\mathrm{First}(\mathrm{E}_i) \cdot_{a_1} e(\nu_{R_1})) \cdot_{a_2} e(\nu_{R_2})) \cdot_{a_3} \cdots) \cdot_{a_m} e(\nu_{R_m})) \tag{1}$$

can be computed in $O(|\,\mathrm{E}\,|)$. Since, this is equivalent to perform disjoint unions. Thus, the set Follow$(\mathrm{E}, \overline{f})$ can be computed in time $O(|\Sigma_0| \cdot |\,\mathrm{E}\,|)$. Therefore, the set of transition rules can be computed in $O(|\Sigma_0| \cdot |\,\mathrm{E}\,| \cdot ||\,\mathrm{E}\,||)$.

From this algorithm the following theorem is obtained.

Theorem 17. *Let* E *be a tree regular expression. The tree automaton* $\mathcal{R}_{\mathrm{E}}$ *associated to* E *is computed in* $O(|\Sigma_0| \cdot |\,\mathrm{E}\,| \cdot ||\,\mathrm{E}\,||)$ *time and space complexity.*

The time complexity of the Algorithm proposed by Dietrich and Meinecke [7] to convert a regular tree expression to its equation automaton is the time complexity of the computation of the sets of Follow(E, f) for $f \in \Sigma$. As the computation of Follow sets can be performed in $O(|\,\mathrm{E}\,| \cdot ||\,\mathrm{E}\,||)$ time, we get the following Corollary.

Corollary 18. *The equation tree automaton of regular tree expression can be computed in* $O(R \cdot |\,\mathrm{E}\,| \cdot ||\,\mathrm{E}\,||)$ *time and space complexity.*

6 Conclusion

In this paper the notion of position tree automaton associated with the regular tree expression has been defined. It is shown that this automaton is the generalization of position automaton introduced by Glushkov. We proved that a regular tree expression E can be converted into a reduced position tree automaton in $O(|\mathrm{E}| \cdot ||\mathrm{E}||)$ time.

By the use of the Algorithm 2 we also proved the time complexity of the Algorithm proposed by Dietrich and Meinecke to construct the equation automaton.

References

1. Antimirov, V.: Partial derivatives of regular expressions and finite automaton constructions. Theoretical Computer Science 155, 291–319 (1996)
2. Bruggemann-Klein, A.: Regular expressions into finite automata. Theoretical Computer Science 120, 197–213 (1993)
3. Chang, C.H., Paige, R.: From regular expressions to DFAs using compressed NFAs. Theoretical Computer Science 178, 136 (1997)
4. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Loding, C., Tison, S., Tommasi, M.: Tree automata techniques and applications (October 2007), `http://www.grappa.univ-lille3.fr/tata`
5. Cortes, C., Haffner, P., Mohri, M.: Rational kernels: Theory and algorithms. Journal of Machine Learning Research 5, 1035–1062 (2004)
6. Glushkov, V.M.: The abstract theory of automata. Russian Mathematical Surveys 16, 1–53 (1961)
7. Kuske, D., Meinecke, I.: Construction of tree automata from regular expressions. RAIRO - Theoretical Informatics and Applications 45, 347–370 (2011)
8. Ouardi, F., Ziadi, D.: Efficient weighted expressions conversion. Informatique Théorique et Application 42(2), 285–307 (2008)
9. Trakhtenbrot, B.: Origins and metamorphoses of the trinity: Logic, nets, automata. In: Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science, pp. 26–29. IEEE Computer Society Press (June 1995)
10. Ziadi, D., Ponty, J.L., Champarnaud, J.M.: Passage d'une expression rationnelle a un automate fini non deterministe. Bulletin of the Belgian Mathematical Society - Simon Stevin 4, 177–203 (1997)

Convergence of Newton's Method over Commutative Semirings[*]

Michael Luttenberger and Maximilian Schlund

Institut für Informatik, Technische Universität München,
Boltzmannstr. 3, 85748 Garching, Germany
{luttenbe,schlund}@model.in.tum.de

Abstract. We give a lower bound on the speed at which Newton's method (as defined in [5,6]) converges over arbitrary ω-continuous commutative semirings. From this result, we deduce that Newton's method converges within a finite number of iterations over any semiring which is "collapsed at some $k \in \mathbb{N}$" (i.e. $k = k + 1$ holds) in the sense of [1]. We apply these results to (1) obtain a generalization of Parikh's theorem, (2) to compute the provenance of Datalog queries, and (3) to analyze weighted pushdown systems. We further show how to compute Newton's method over any ω-continuous semiring.

1 Introduction

Fixed-point iteration is a standard approach for solving equation systems of the form $\boldsymbol{X} = F(\boldsymbol{X})$: The naive approach is to compute the sequence $\boldsymbol{X}_{i+1} = F(\boldsymbol{X}_i)$ given some suitable initial approximation $\boldsymbol{X}_0$. In calculus Banach's fixed-point theorem guarantees that the constructed sequence converges to a solution if F is a contraction over a complete metric space. In computer science, Kleene's fixed-point theorem[1] guarantees convergence if F is an ω-continuous map over a complete partial order. In reference to Kleene's fixed-point theorem, we will call the naive application of fixed-point iteration "Kleene's method" in the following. It is well-known that Kleene's method converges only very slowly in general. Consider the equation $X = 1/2X^2 + 1/2$ over the reals. Kleene's method $\boldsymbol{\kappa}^{(h+1)} = 1/2(\boldsymbol{\kappa}^{(h)})^2 + 1/2$ converges from below to the only solution $x = 1$ starting from the initial approximation $\boldsymbol{\kappa}^{(0)} = 0$. However, it takes 2^{h-3} iterations to gain h bits of precision, i.e. $1 - \boldsymbol{\kappa}^{(2^{h-3})} \leq 2^{-h}$ [8].

Therefore, many approximation schemes do not apply Kleene's method, instead they construct from F a new map G to which fixed-point iteration is then applied: Newton's method, for instance, obtains G from a nonlinear function F by linearization. In above example, $F(X) = 1/2X^2 + 1/2$ is replaced by $G(X) = 1/2X + 1/2$ yielding the sequence $\boldsymbol{\nu}^{(h+1)} = G(\boldsymbol{\nu}^{(h)}) = 1 - 2^{-h}$ for $\boldsymbol{\nu}^{(0)} = 0$, i.e. we get one bit of precision with each iteration.

[*] This work was partially funded by the DFG project "Polynomial Systems on Semirings: Foundations, Algorithms, Applications".

[1] Depending on literature, this result is also attributed to Tarski.

A system $\boldsymbol{X} = F(\boldsymbol{X})$ where F is given in terms of polynomials over a semiring is called algebraic. In computer science, algebraic systems arise e.g. in the analysis of procedural programs where their least solution describes the set of runs of the program (possibly evaluated under a suitable abstraction). Motivated by the fast convergence of Newton's method over the reals, in [5,6] (see [7] for an updated version) Newton's method was extended to algebraic systems over ω-continuous semirings: It was shown there that Newton's method always converges monotonically from below to the least solution at least as fast as Kleene's method. In particular, there are semirings where Newton's method converges within a finite number of iterations while Kleene's method does not. This extension of Newton's method found several applications in verification (see e.g. [7,4,11]). Independent of the mentioned work, the same extension of Newton's method has been proposed in [17] in the setting of combinatorics which led to new efficient algorithms for random generation of objects.

In this article we give a lower bound on the speed at which Newton's method converges over arbitrary *commutative* ω-continuous semirings. We measure the speed by essentially looking at the number of terms evaluated by Newton's method. To make this more precise, consider the equation $X = aX^2 + c$ in the formal parameters a, c (e.g. over the semiring of formal power series). Its least solution is the series $B = \sum_{n\in\mathbb{N}} C_n a^n c^{n+1}$ with C_n the n-th Catalan number.

The Kleene approximations $\boldsymbol{\kappa}^{(h+1)} := a\boldsymbol{\kappa}^{(h)}\boldsymbol{\kappa}^{(h)} + c$ of B are always polynomials and one can show that the number of correctly computed coefficients increases by one in each iteration, e.g. $\boldsymbol{\kappa}^{(3)} = c + ac^2 + 2a^2c^3 + a^3c^4$. By contrast, the Newton approximations $\boldsymbol{\nu}^{(h)}$ are (infinite) power series. It follows easily from the characterization [5] of the Newton approximations by "tree-dimension" (see Sec. 3), that the coefficient of $a^n c^{n+1}$ in $\boldsymbol{\nu}^{(h)}$ has converged to C_n if and only if $n + 1 < 2^h$, i.e. the number of coefficients which have converged is now roughly doubled in each iteration. In [17] this property is called *quadratic convergence* (see also Ex. 5) and is used there to argue that Newton's method allows to efficiently compute a finite number of coefficients of the formal power series representing a generating function.

In programs analysis, monomials correspond to runs of a program and we are in general not only interested in the coefficients of a finite number of monomials. We show in Theorem 6 for *any* monomial m that either its coefficient in $\boldsymbol{\nu}^{(n+k+1)}$ has already converged or it is bounded from below by 2^{2^k} (where n is the number of variables of the given algebraic system). In particular, if the coefficient of m is less than 2^{2^k} in $\boldsymbol{\nu}^{(n+k+1)}$, then we know that it has converged. Using this theorem, we extend Parikh's theorem[2] to multiplicities bounded by a given $k \in \mathbb{N}$ (see Sec. 5.1). From this it follows that the set of monomials whose coefficients have converged in the h-th Newton approximation is Presburger definable. In Sec. 5.2 we apply these results to the problem of computing the provenance of a Datalog query improving on the algorithms proposed in [12]. As a further application of our results, we show in Sec. 5.3 how Newton's method by virtue of Theorem

[2] Parikh's theorem states that the commutative image of a context-free grammar is a semilinear set, i.e. definable by means of a Presburger formula.

6 can be used to speed up the computation of predecessors and successors in weighted pushdown-systems [18] which has applications e.g. in the analysis of procedural programs or generalized authorization problems in SPKI/SDSI. As a side result, we also show how to compute Newton's method for algebraic systems over arbitrary, also noncommutative, ω-continuous semirings (Sec. 3, Definition 2). Due to the page limit, we refer the reader to the technical report [13] for the missing proofs.

2 Preliminaries

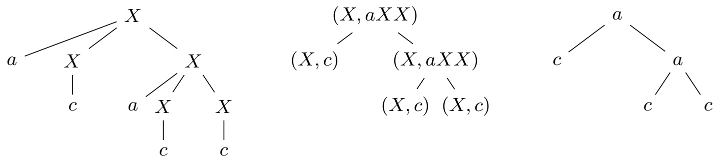
$\mathbb{N}$ denotes the nonnegative integers (natural numbers). $\mathbb{N}_\infty$ are the natural numbers extended by a greatest element ∞. For $k \in \mathbb{N}$ let $\mathbb{N}_k = \{0, 1, \ldots, k\}$. A^* ($\mathsf{A}^\oplus$) denotes the free (commutative) monoid generated by A. Elements of $\mathsf{A}^\oplus$ are usually written as monomials (in the variables A). $\mathbb{N}_\infty\langle\langle \mathsf{A}^* \rangle\rangle$ denotes the set of all total functions from A^* to $\mathbb{N}_\infty$. These functions are commonly represented a formal power series (in noncommuting variables A and coefficients in $\mathbb{N}_\infty$). Analogously for $\mathbb{N}_\infty\langle\langle \mathsf{A}^\oplus \rangle\rangle$ with now commuting variables.

A context-free grammar is a triple $G = (\mathcal{X}, \mathsf{A}, R)$ with variables (nonterminals) $\mathcal{X}$, alphabet (formal parameters) A, and (rewrite) rules R. We do not assume a specific start symbol. G is nonexpansive if no variable $X \in \mathcal{X}$ can be rewritten into a sentential form in which X occurs at least twice (see e.g. [19]). G is in quadratic normal form if any rule $X \to u_0 X_1 u_1 \ldots u_{r-1} X_r u_r$ of G satisfies $u_0 u_1 \ldots u_r \in \mathsf{A}^+$, $X_1 X_2 \ldots X_r \in \mathcal{X}^+$, and $r \in \{0, 2\}$.

We slightly deviate from the standard representation of derivation trees: We label the nodes of a derivation tree directly by the corresponding rule (see Example 1). For $X \in \mathcal{X}$, a derivation tree of G is an X-tree if its root is labeled by a rule rewriting X. The word represented by a derivation tree is called its yield. The ambiguity of a context-free grammar G w.r.t. to $X \in \mathcal{X}$ is the map $\mathsf{amb}_X \in \mathbb{N}_\infty\langle\langle \mathsf{A}^* \rangle\rangle$ which assigns to a word $w \in \mathsf{A}^*$ the number of X-trees of G which yield w. Analogously, we define the *commutative* ambiguity $\mathsf{camb}_X \in \mathbb{N}_\infty\langle\langle \mathsf{A}^\oplus \rangle\rangle$ which assigns to each monomial $m \in \mathsf{A}^\oplus$ the number of X-trees of G which yield a permutation of m. G is unambiguous w.r.t. X if every word has a unique X-tree, i.e. if amb_X takes only values in $\{0, 1\}$.

The family $\mathsf{amb} = (\mathsf{amb}_X \mid X \in \mathcal{X})$ can equally be characterized as the least solution of the algebraic system $\boldsymbol{X} = F_G(\boldsymbol{X})$ over $\mathbb{N}_\infty\langle\langle \mathsf{A}^* \rangle\rangle$ consisting of the equations $X = \sum_{(X,\gamma)\in P} \gamma$. In particular, for any interpretation $\iota\colon \mathsf{A} \to S$ of the alphabet symbols as elements of some ω-continuous semiring $\langle S, +, \cdot \rangle$ it is known [3,7] that amb evaluates under (the ω-continuous homomorphism induced by) ι to the least solution of the algebraic system $\boldsymbol{X} = F^\iota(\boldsymbol{X})$ over S where F^ι is obtained from F by substituting every occurrence of $a \in \mathsf{A}$ by $\iota(a)$. Similarly, any approximation scheme for amb translates to an approximation scheme for $\iota(\mathsf{amb})$ over S. As we can associate with any algebraic system $\boldsymbol{X} = F(\boldsymbol{X})$ over $\langle S, +, \cdot \rangle$ a context-free grammar (in the restricted from defined above) such that $\boldsymbol{X} = F_G(\boldsymbol{X})$ has the same least solution, it suffices to study how to approximate amb. Analogously for a commutative semiring $\langle S, +, \cdot \rangle$ and camb. We therefore do not introduce ω-continuous semirings and algebraic systems formally, but refer the reader to e.g. [19].

Example 1. Consider the grammar $G_L \colon X \to aXX \mid c$. The language $L(G_L)$ generated by G_L is known as Lukasiewicz language of all proper[3] binary trees with binary nodes labeled by a, and leaves labeled by c represented as a word using Polish notation. Below on the left the common depiction of the derivation tree of *acacc* is shown; the middle tree is the representation used in the following which is isomorphic to the binary tree represented by *acacc* shown on the right:



As G_L is unambiguous, amb enumerates all proper binary trees. camb on the other hand is the generating function of proper binary trees, i.e. $\mathsf{camb}(a^n c^{n+1})$ is the n-th Catalan number C_n.

$$\mathsf{camb} = c + ac^2 + 2a^2c^3 + 5a^3c^4 + 14a^4c^5 + 42a^5c^6 + 132a^6c^7 + 429a^7c^8 + 1430a^8c^9 + \dots$$

3 Newton's Method for Context-Free Grammars

The Kleene approximation $\boldsymbol{\kappa}^{(h)}$ of amb ($\boldsymbol{\kappa}^{(h+1)} = F_G(\boldsymbol{\kappa}^{(h)})$ with $\boldsymbol{\kappa}^{(0)} = 0$) can be characterized by means of the derivation trees evaluated by them (see e.g. [5]): The X-component $\boldsymbol{\kappa}_X^{(h)}$ of $\boldsymbol{\kappa}^{(h)}$ assigns to $w \in \mathsf{A}^*$ the number of X-trees of *height less than h* which yield w. In [6,5] the notion of dimension was introduced to give a similar characterization of the Newton approximations $\boldsymbol{\nu}^{(h)}$: The *dimension* of a (rooted) tree t is the maximal height of any perfect[4] binary tree which is a minor of t. The dimension is also known as Horton-Strahler number or register number [9]. Then $\boldsymbol{\nu}_X^{(h)}$ assigns to $w \in \mathsf{A}^*$ the number of X-trees *of dimension less than h* which yield w. Analogously for camb. We use this result to unfold any context-free grammar G w.r.t. to the dimension into a new context-free grammar $G^{(h)}$ so that the (commutative) ambiguity of $G^{(h)}$ is exactly the h-th Newton approximation of the (commutative) ambiguity of G. One advantage of this new definition is that it allows to effectively compute Newton's method over any ω-continuous semiring for which we can compute the semiring operations and the Kleene star. By contrast, the algebraic definition in [6,5] requires the user to find in every iteration step a certain semiring element. There, only for particular semirings, e.g. when addition is idempotent, it was shown how to construct these elements. For the unfolding we assume that G is in quadratic normal form. This is no real restriction but simplifies the presentation.[5]

[3] A binary tree is proper if every node is either binary or nullary.

[4] A proper binary tree is perfect if every leaf has the same distance to the root.

[5] See the technical report [13] for how to unfold arbitrary context-free grammars.

Definition 2. *Let G be a context-free grammar $G = (\mathcal{X}, \mathsf{A}, R)$. Set $\mathcal{X}^\nu := \{X^{(d)}, \hat{X}^{(d)} \mid X \in \mathcal{X}, d \in \mathbb{N}\}$. The unfolding $G^\nu = (\mathcal{X}^\nu, \mathsf{A}, R^\nu)$ of G is:*

- *$X^{(d)} \to \hat{X}^{(e)}$ for every $d \in \mathbb{N}$, and every $0 \leq e < d$.*
- *If $X \to u_0$ in R, then $\hat{X}^{(0)} \to u_0$.*
- *If $X \to_G u_0 X_1 u_1 X_2 u_2$ in R, then for every $d \geq 1$:*

$$\begin{aligned}\hat{X}^{(d)} &\to u_0 X^{(d)} u_1 \hat{X}^{(d)} u_2\\ \hat{X}^{(d)} &\to u_0 \hat{X}^{(d)} u_1 X^{(d)} u_2\\ \hat{X}^{(d)} &\to u_0 \hat{X}^{(d-1)} u_1 \hat{X}^{(d-1)} u_2.\end{aligned}$$

For any given $h \in \mathbb{N}$ let $G^{(h)} = (\mathcal{X}^{(h)}, \mathsf{A}, R^{(h)})$ be the context-free grammar induced by the variables $\{X^{(h)} \mid X \in \mathcal{X}\}$. The h-th Newton approximation $\boldsymbol{\nu}_X^{(h)}$ of the (commutative) ambiguity of G w.r.t. X is the (commutative) ambiguity of $G^{(h)}$ w.r.t. $X^{(h)}$.

Lemma 3. *Every $\hat{X}^{(d)}$-tree ($X^{(d)}$-tree) has dimension exactly (less than) d. There is a yield-preserving bijection between the $\hat{X}^{(d)}$-trees ($X^{(d)}$-trees) and the X-trees of dimension exactly (less than) d.*

Newton's method is closely related to nonexpansive grammars and related notions like quasi-rational languages:

Theorem 4. *Let $G = (\mathcal{X}, \mathsf{A}, R)$ be a context-free grammar.*

1. *All Newton approximations of* camb *are rational in $\mathbb{N}_\infty\langle\langle \mathsf{A}^\oplus \rangle\rangle$.*
2. *Newton's method converges to* amb *(*camb*) of G within a finite number of iterations if and only if G is nonexpansive. If G is nonexpansive, then Newton's method converges within $|\mathcal{X}|$ iterations.*

If G is expansive, not much can be said regarding convergence speed in the noncommutative setting as illustrated by any unambiguous grammar G: For a given $w \in L(G)$, the least h with $\boldsymbol{\nu}_X^{(h)}(w) = \mathsf{amb}_X(w)$ is simply the dimension of the unique X-tree yielding w. Thus, in the following section we focus on the commutative setting and study the speed at which Newton's method converges to camb by means of a lower bound on all coefficients which have not yet converged.

Example 5. Unfolding G_L (see Ex. 5) w.r.t. the dimension gives us $\hat{X}^{(0)} \to c$, $X^{(1)} \to \hat{X}^{(0)}$ and for $d > 0$

$$\begin{aligned}X^{(d)} &\to \hat{X}^{(0)} \mid \hat{X}^{(1)} \mid \ldots \mid \hat{X}^{(d-1)}\\ \hat{X}^{(d)} &\to aX^{(d)}\hat{X}^{(d)} \mid a\hat{X}^{(d)}X^{(d)} \mid a\hat{X}^{(d-1)}\hat{X}^{(d-1)}\end{aligned}$$

Modulo commutativity, we can deduce from this the following rational expressions for the first few approximations of camb: $\boldsymbol{\nu}^{(0)} = 0$, $\boldsymbol{\nu}^{(1)} = c$,

$$\begin{aligned}\boldsymbol{\nu}^{(2)} &= (2ac)^* ac^2 + c\\ &= c + ac^2 + 2a^2c^3 + 4a^3c^4 + \ldots\\ \boldsymbol{\nu}^{(3)} &= (2a((2ac)^*ac^2 + c))^* a((2ac)^*ac^2)^2\\ &= c + ac^2 + 2a^2c^3 + 5a^3c^4 + 14a^4c^5 + 42a^5c^6 + 132a^6c^7 + 428a^7c^8 + \ldots\end{aligned}$$

We have expanded the series until the first coefficient which differs from camb (see Ex. 1) to exemplify the notion of quadratic convergence introduced in [17]: $\boldsymbol{\nu}^{(h)}$ differs from camb in the coefficient of $a^n c^{n+1}$ if and only if $n+1 \geq 2^h$ as any tree with less than 2^h leaves can only have dimension at most $h-1$. This also shows that Newton's method cannot converge faster than quadratic in this sense. Note that although Newton's method converges quadratically w.r.t. camb, it only converges linearly over the reals: Consider G_L interpreted as an algebraic system over $\mathbb{R}$ with $\iota(a) = \iota(c) = 1/2$ yielding $X = 1/2X^2 + 1/2$. By also reading the unfolded grammar as an algebraic system and interpreting the alphabet by the same ι we recover the Newton approximations over $\mathbb{R}$: $X^{(0)} = 0$, $\hat{X}^{(0)} = 1/2$, and for $d > 0$:

$$X^{(d)} = X^{(d-1)} + \hat{X}^{(d-1)} \text{ and } \hat{X}^{(d)} = (1 - X^{(d)})^{-1} \cdot 1/2 \left(\hat{X}^{(d-1)}\right)^2$$

Induction shows that indeed $\iota(\boldsymbol{\nu}^{(h)}) = X^{(h)} = 1 - 2^{-h}$.

4 Rate of Convergence Modulo Commutativity

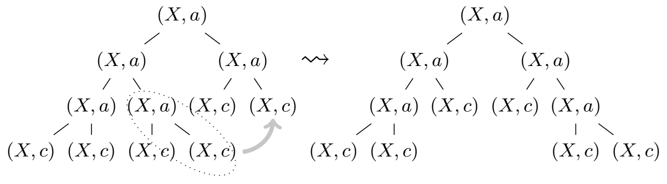
Let $G = (\mathcal{X}, \mathsf{A}, R)$ be a context-free grammar. In the following n denotes $|\mathcal{X}|$ and $\boldsymbol{\nu}^{(h)}$ denotes the h-th Newton approximation of camb of G, i.e. $\boldsymbol{\nu}_X^{(h)} = \mathsf{camb}_{X^{(h-1)}}$. We say that two X-trees (w.r.t. G) are *Parikh-equivalent* if they yield the same word up to commutativity. We show that after $n+1$ iterations all coefficients which have not converged yet are bounded from below by 2^{2^k}.

Theorem 6. *For all $k \geq 0$ and $\boldsymbol{v} \in \mathsf{A}^{\oplus}$: $\boldsymbol{\nu}_X^{(n+k+1)}(\boldsymbol{v}) \geq \min(\mathsf{camb}_X(\boldsymbol{v}), 2^{2^k})$.*

Proof (sketch). Assume there is $\boldsymbol{v} \in \mathsf{A}^{\oplus}$ with $\boldsymbol{\nu}_X^{(n+k)}(\boldsymbol{v}) < \mathsf{camb}_X(\boldsymbol{v})$. This means there exists some derivation tree t with dimension $\mathsf{dim}(t) \geq n+k+1$ and yield $\boldsymbol{v}$ modulo commutativity. Essentially we show that t witnesses the existence of at least 2^{2^k} different, but Parikh-equivalent trees of lower dimension.

To make this more precise, we need to introduce $l(t)$: Recall that we labeled the nodes of derivation trees by rules of G. A variable Y is a label of t if there is at least one node which is labeled by a rule rewriting Y. Then $l(t)$ is the number of variables labeling t. We prove by induction on the number of vertices of t that if $\mathsf{dim}(t) \geq l(t) + k + 1$, then there exist at least 2^{2^k} Parikh-equivalent trees of dimension at most $l(t) + k$.

Assume that t has dimension $l(t)+k+1$ and exactly two subtrees t_1, t_2 having dimension exactly $l(t) + k$ and furthermore $l(t_1) = l(t_2) = l(t)$ (all other cases reduce to this one, or follow from the induction hypothesis). Since t_1 has dimension $l(t) + k$ it contains a perfect binary tree of height $l(t) + k$ as a minor. The set of nodes of this minor on level k define 2^k (independent) subtrees of t_1. Each of these 2^k subtrees has height at least $l(t)$, and thus by the Pigeonhole principle contains a path with two variables repeating. We call the partial derivation tree defined by these two repeating variables a *pump-tree*. We relocate any subset of these 2^k pump-trees to t_2 which is possible since $l(t_2) = l(t) = l(t_1)$. See the following picture for an illustration of the relocation process (we have two choices for the pump-tree on the left, yielding four possible "remainders").



Each of these 2^{2^k} choices produces a different tree $\tilde{t}$—the trees differ in the subtree $\tilde{t_1}$. We now apply the following result from [6]: For every derivation tree t there is a Parikh-equivalent tree $\tilde{t}$ of dimension at most $l(t)$. Applying this result to $\tilde{t_2}$ allows us to reduce the dimension of each $\tilde{t}$ to at most $\mathsf{dim}(t_1) = l(t) + k$. This way we obtain at least 2^{2^k} different Parikh-equivalent trees of dimension at most $\mathsf{dim}(t_1) = l(t) + k$.

Remark 7. As we can also choose t_2 as the source and t_1 as the destination of the relocation process, we obtain in fact a lower bound of 2^{1+2^k}, which is best possible (in this form): Looking at Ex. 5 for $k = 0$ we obtain a lower bound of $\boldsymbol{\nu}^{(2)}(v) \geq 4$ for all coefficients that have not converged yet – and indeed $\boldsymbol{\nu}^{(2)}(a^3c^4) = 4$.

It would be nice to have a non-uniform global bound on the coefficients $\boldsymbol{\nu}^{(n+1+k)}(v)$ (i.e. some bound that depends on k and $|v|$). However, the following grammar H shows that this cannot be done without taking into account the structure of the grammar: $H : Y \to BY \mid BX, B \to b, X \to aXX \mid c$. This grammar contains G_L, but any word produced by Y can have an arbitrarily long prefix of b's and each such prefix has a unique derivation. Thus $\mathsf{camb}_Y(b^m a^n c^{n+1}) = \mathsf{camb}_X(a^n c^{n+1}) = C_n$.

We say that a ω-continuous semiring S is *collapsed* at some positive integer k if in S the identity $k = k + 1$ holds (see e.g. [1]). For instance, the semirings $\mathbb{N}_k\langle\langle \mathsf{A}^* \rangle\rangle$ and $\mathbb{N}_k\langle\langle \mathsf{A}^\oplus \rangle\rangle$ are collapsed at k. For $k = 1$ the semiring is idempotent.

Corollary 8. *Newton's method converges within $n + \log\log k$ iterations for any algebraic system with n variables over a commutative semiring collapsed at k.*

5 Applications

5.1 Parikh's Theorem for Bounded Multiplicities

Petre [15] defines a hierarchy of power series over $\mathbb{N}_\infty\langle\langle \mathsf{A}^\oplus \rangle\rangle$ and showed that this hierarchy is strict. In particular he shows that Parikh's Theorem does not hold if multiplicities are considered. Here we combine our convergence result and some identities for weighted rational expressions over commutative k-collapsed semirings to show that moving from $\mathbb{N}_\infty\langle\langle \mathsf{A}^\oplus \rangle\rangle$ to $\mathbb{N}_k\langle\langle \mathsf{A}^\oplus \rangle\rangle$ allows us to prove a Parikh-like theorem, i.e. we give a semilinear characterization of camb_G.

In the following, let k denote a fixed positive integer. By Theorem 4 and Corollary 8 we know that camb_G is rational modulo $k = k+1$. In the idempotent setting ($k = 1$), see e.g. [16] the identities (i) $(x^*)^* = x^*$, (ii) $(x+y)^* = x^*y^*$, and (iii) $(xy^*)^* = 1 + xx^*y^*$ can be used to transform any regular expression into a regular expression in "semilinear normal form" $\sum_{i=1}^{r} w_{i,0}w_{i,1}^* \dots w_{i,l_r}^*$ with $w_{i,j} \in \mathsf{A}^*$. It is not hard to deduce the following identities over $\mathbb{N}_k\langle\langle \mathsf{A}^\oplus \rangle\rangle$ where $x^{<r}$ abbreviates the sum $\sum_{i=0}^{r-1} x^i$. By $\mathsf{supp}(x)$ we denote the characteristic series of the support of x:

Lemma 9. *The following identities hold over* $\mathbb{N}_k\langle\langle \mathsf{A}^\oplus \rangle\rangle$*:*

$$\begin{aligned}
&(\mathsf{I1}) \quad kx &&= k\,\mathsf{supp}(x)\\
&(\mathsf{I2}) \quad (\gamma x)^* &&= (\gamma x)^{<\lceil \log_\gamma k \rceil} + kx^{\lceil \log_\gamma k \rceil} x^*\\
&(\mathsf{I3}) \quad (x^*)^* &&= kx^*\\
&(\mathsf{I4}) \quad (x+y)^* &&= (x+y)^{<k} + x^k x^* + y^k y^* + kxy(x+y)^{\max(k-2,0)}x^*y^*\\
&(\mathsf{I5}) \quad (xy^*)^* &&= 1 + xy^* + x^2x^* + x^2y\textstyle\sum_{0\le m,j<k-2}\binom{2+m+j}{1+j}x^m y^j\\
& && \quad + kx^2y(x^{\max(k-2,0)} + y^{\max(k-2,0)})x^*y^*
\end{aligned}$$

for γ any integer greater than one.

Consider a rational series $\mathfrak{r} \in \mathbb{N}_k\langle\langle \mathsf{A}^\oplus \rangle\rangle$ represented by the rational expression ρ. The above identities, where $(\mathsf{I3}), (\mathsf{I4}), (\mathsf{I5})$ generalizes (i), (ii), (iii), respectively, allow us to reduce the star height of ρ to at most one by distributing the Kleene stars over sums and products yielding a rational expression ρ' of the form $\rho' = \sum_{i=1}^{s} \gamma_i w_{i,0} w_{i,1}^* \dots w_{i,l_i}^*$ ($w_{i,j} \in \mathsf{A}^*, \gamma_i \in \mathbb{N}_k$) which still represents $\mathfrak{r}$ over $\mathbb{N}_k\langle\langle \mathsf{A}^\oplus \rangle\rangle$. By $(\mathsf{I1})$ we know that, if $\gamma_{i,0} = k$, we may replace $w_{i,0}w_{i,1}^* \dots w_{i,l_i}^*$ by its support which is a linear set in $\mathbb{N}^\mathsf{A}$.

Theorem 10. *Every rational* $\mathfrak{r} \in \mathbb{N}_k\langle\langle \mathsf{A}^\oplus \rangle\rangle$ *can be represented as a finite sum of weighted linear sets, i.e.* $\mathfrak{r} = \sum_{i\in[s]} \gamma_i\, \mathsf{supp}(w_{i,0}w_{i,1}^* \dots w_{i,l}^*)$ *with* $w_{i,j} \in \mathsf{A}^*$ *and* $\gamma_i \in \mathbb{N}_k$*.*

Example 11. The rational expression $\rho = (a + 2b)^*$ represents the power series $\sum_{i,j\in\mathbb{N}} 2^j a^i b^j$ in $\mathbb{N}_\infty\langle\langle \mathsf{A}^\oplus \rangle\rangle$. Computing over $\mathbb{N}_2\langle\langle \mathsf{A}^\oplus \rangle\rangle$ we may transform ρ as follows:

$(a+2b)^* \overset{(\mathsf{I4})}{=} (a+2b)^{<2} + a^2a^* + (2b)^2(2b)^* + 2a(2b)a^*(2b)^* = a^* + 2(bb^* + aba^*b^*) = a^* + 2(bb^*a^*) \overset{(\mathsf{I1})}{=} 1\,\mathsf{supp}(a^*) + 2\,\mathsf{supp}(bb^*a^*)$

Corollary 12. *For every $k \in \mathbb{N}_\infty$ we can construct a formula of Presburger arithmetic that represents the set* $\{\boldsymbol{v} \in \mathbb{N}^\mathsf{A} \mid \mathsf{camb}_{G,X}(\boldsymbol{v}) = k\}$*.*

This corollary can be applied to inclusion testing between two rational series over $\mathbb{N}_k\langle\langle \mathsf{A}^\oplus \rangle\rangle$ which is relevant e.g. for detecting early convergence of Newton's method, i.e. if $\boldsymbol{\nu}^{(h+1)} = \boldsymbol{\nu}^{(h)}$. Although we know that after $n + \log\log k$ steps the method has converged, in applications (see Sec. 5.2) n could be quite large and the $n + \log\log k$ bound might be too pessimistic.

5.2 Provenance Computation for Datalog

Roughly speaking, provenance is additional information attached to the results of a database query explaining how said results were obtained from the current facts in the database. Provenance information is important e.g. to implement updatable views [10]. Recently, commutative ω-continuous semirings were proposed as provenance annotations where the provenance of unions or projections is modelled by addition of the annotation and joins yield multiplications. Tagging the tuples from the facts in the database allows us to trace back the provenance of the results by solving an algebraic system [12].

For an example consider the binary relation E depicted below (first table). The Datalog query $T(x,y)$:- $E(x,y)$; $T(x,y)$:- $E(x,z), E(z,y)$ computes its transitive closure $T = E^*$ (second table).

X	Y	
a	b	e_1
b	b	e_2
b	c	e_3
c	d	e_4

X	Y	
a	b	$X_1 = e_1 \quad + X_1X_4$
a	c	$X_2 = X_1X_5$
a	d	$X_3 = X_1X_6 + X_2X_7$
b	b	$X_4 = e_2 \quad + X_4X_4$
b	c	$X_5 = e_3 \quad + X_4X_5$
b	d	$X_6 = X_4X_6 + X_5X_7$
c	d	$X_7 = e_4$

$$\left.\begin{aligned} X_1 &= X_4^* e_1 \\ X_2 &= (X_4^*)^2 e_1 e_3 \\ X_3 &= [(X_4^*)^2 + (X_4^*)^3] e_1 e_3 e_4 \\ X_4 &= \textstyle\sum_{n\geq 0} C_n (e_2)^{n+1} \\ X_5 &= X_4^* e_3 \\ X_6 &= (X_4^*)^2 e_3 e_4 \\ X_7 &= e_4 \end{aligned}\right\} \overset{(1=1+1)}{=} \begin{array}{c} e_1 e_2^* \\ e_1 e_2^* e_3 \\ e_1 e_2^* e_3 e_4 \\ e_2 e_2^* \\ e_3 e_2^* \\ e_2^* e_3 e_4 \\ e_4 \end{array}$$

To capture the so called "how-provenance" we tag every tuple in E by a letter from $\Sigma = \{e_1, e_2, e_3, e_4\}$. The provenance of the k-th tuple in T is the value of X_k in the (least) solution (over a suitable semiring) of the algebraic system representing the query. In our example the solution over $\mathbb{N}\langle\!\langle \mathsf{A}^{\oplus} \rangle\!\rangle$ can be computed by hand and we can also give a very short representation as rational expressions if we assume idempotence of addition $(1 = 1 + 1)$. From the result we can see that the tuple (b,d) can be obtained by a join of (b,c) and (c,d), preceded by any number of joins of (b,b) with itself [6].

Depending on our choice of the semiring we obtain a coarser or finer view on the provenance. As $\mathbb{N}_\infty\langle\!\langle \mathsf{A}^{\oplus} \rangle\!\rangle$ is the commutative semiring, freely generated by A, we can regard it as the universal provenance semiring [12]. However, $\mathbb{N}_\infty\langle\!\langle \mathsf{A}^{\oplus} \rangle\!\rangle$ is in some sense a bad choice for representing solutions, as we cannot do this finitely. Green et al. [12] therefore resort to compute the complete provenance series only if it is finite by enumerating all derivation trees using Kleene's method essentially; if the power series is an infinite sum they only compute the coefficient for a given monomial.

For many applications, idempotent semirings suffice to capture interesting provenance information. Useful examples are the tropical semiring $\langle \mathbb{N}_\infty, \min, + \rangle$, or the Viterbi-semiring $\langle [0,1], \max, \cdot \rangle$ for probabilistic settings. [12] raised the open question how to compute provenance over the tropical semiring, which can be done by Newton's method as already described in [6]. A useful generalization which is not idempotent is the k-tropical semiring $\mathcal{T}_k$ [14] which was used there for general k-shortest distance computations. This semiring satisfies the identity

[6] More precisely, the operations are joins followed by projections.

$k = k+1$, so by our results Newton's method can be used to calculate provenance series over $\mathcal{T}_k$ in $n + \log\log k$ steps.

As already remarked in [12], idempotent semirings are often too coarse an abstraction in a database context where one often considers the so called *bag-semantics* (i.e. we also care about the multiplicities of query results or provenance information). The k-collapsed semirings $\mathbb{N}_k\langle\langle \mathsf{A}^{\oplus}\rangle\rangle$ are a possible way out of the dilemma that we want to capture the bag-semantics to some extent but cannot use the most general semiring $\mathbb{N}_\infty\langle\langle \mathsf{A}^{\oplus}\rangle\rangle$ since its elements are not finitely representable in general. Suppose, we want to compute provenance for a recursive query and are satisfied with a power series having coefficients less than $k = 2^{64}$ (i.e. standard 64-bit integers). By Theorem 6 we know that Newton's method converges after at most $n + 6$ steps.

5.3 Analysis of Weighted Pushdown Systems

A Pushdown system (PDS) $\langle Q, \Gamma, \Delta\rangle$ consists of a finite set of control states Q, a finite set of stack symbols Γ, and a set of rewrite rules $\Delta \subset Q\Gamma \to Q\Gamma^{\leq 2}$. A PDS induces an infinite graph over the $Q\Gamma^*$ of configurations: there is an edge from $q\gamma$ to $q'\gamma'$ if there is a rule $qA \to q'\rho \in \Delta$ such that $\gamma' = A\gamma''$ and $\gamma' = \rho\gamma''$. In a weighted PDS each rule carries also as weight an element of a semiring $\langle S, +, \cdot\rangle$. The semiring multiplication is used to *extend* weights from single rules to paths, while addition is used to *combine* the weight of several paths. Such weighted graphs arise e.g. in the analysis of procedural programs [18] or in authorization problems [20]. A central problem is: given a configuration c of the graph, determine for any other configuration c' the weight of all finite paths leading from c to c'.

To solve this problem for arbitrary configurations, one builds a weighted finite automaton whose transitions corresponds to particular runs starting in a configuration pA with a single stack symbol and ending in a configuration $q\varepsilon$ with empty stack. The total weight of these paths is the least solution of an algebraic system over the given semiring S. In the standard approach [18] this algebraic system is solved on the fly while constructing the automaton. For this a work list variant of Kleene's method is used. This approach therefore only works for certain semirings and its running time is directly proportional to the number of iterations needed by Kleene's method to converge which depends on the given semiring. Alternatively, as discussed in [2], one can first build the unweighted automaton, and then solve the algebraic system explicitly. We give an example how Newton's method in combination with Theorem 6 allows to speed this up:

Consider the PDS $pA \xrightarrow{a} pAA$, $pA \xrightarrow{b} q$, and $qA \xrightarrow{c} p$ where we have assigned a unique label (weight) to each rule. The PDS encodes a program which always starts in the configuration pA, and we expect it to terminate in $p\varepsilon$. Termination in configuration $q\varepsilon$ is considered to be an error. To simplify debugging, we would like to have, say the k paths from pa to $q\varepsilon$, in particular, these paths should be short. All paths from pa to $p\varepsilon$ resp. $q\varepsilon$ are described by the grammar

$$X \to aXX \mid aYc \text{ and } Y \to aXY \mid b.$$

We first determine the length of the k shortest paths. To this end, we can collapse the alphabet to a singleton, say $\iota(a) = \iota(b) = \iota(c) = z$, and compute the commutative ambiguity of the resulting grammar modulo $k = k + 1$. The coefficient of z^i in camb_X resp. camb_Y then tells us, how many paths (up to k) of length i lead from pA to $p\varepsilon$ resp. $q\varepsilon$. For simplicity, assume $k = 4$. By virtue of Theorem 6 we know that at most $n + 1 + \log\log k = 4$ Newton iterations suffice to compute camb modulo $k = k + 1$. (For comparison, Kleene's method can take up to $\mathcal{O}(k)$ iterations, consider e.g. $pA \to pAA, pA \to qA, qA \to q\varepsilon$.) This gives us: $\mathsf{camb}_X = z^3 + 2z^7 + 2z^{11} + \mathcal{O}(z^{12})$ and $\mathsf{camb}_Y = z + z^5 + 3z^9 + \mathcal{O}(z^{10})$. The partial expansion of camb_Y tells us the four shortest paths from pA to $q\varepsilon$ consist of one path of length 1, one path of length 5, and two paths of length 9 each. For constructing the actual paths, these lengths allows us to early discard paths which cannot contribute to the k shortest paths. For instance, we can now apply Kleene's method and discard after each iteration any path of length at least 10. This will take 5 iterations until we have discovered enough paths.

On the other hand, by virtue of Theorem 6 we know that we discover a sufficient number of paths of any given length l when considering only derivation trees of low dimension. Consider e.g. the restriction of the grammar to derivation trees of dimension at most one (see Def. 2). Dimension 0 gives us the shortest path b from pA to $q\varepsilon$. The unfolding of the grammar to dimension exactly 1 is:

$$\begin{aligned} \hat{X}^{(1)} &\to a\hat{X}^{(1)}abc \mid aabc\hat{X}^{(1)} \mid aabcabc \mid a\hat{Y}^{(1)}c \\ \hat{Y}^{(1)} &\to a\hat{X}^{(1)}b \mid aabc\hat{Y}^{(1)} \mid aabcb \end{aligned}$$

Applying Kleene iteration now to this unfolded grammar, we only enumerate trees of dimension 1 with at most 9 leaves. Within two iterations we obtain enough paths, namely $aabcb$, $(aabc)^2b$, $aaaabcbcb$, and $aaabcabcb$, to answer the query. Note that a path of the form $(aabc)^hb$ has a derivation tree of dimension 1, but of height $h+1$, i.e. it takes $h+1$ Kleene iterations on the original grammar to discover this path. By increasing k, the gap between Newton's method and Kleene's method can thus be made arbitrarily large.

6 Future Work

For proper binary trees, [9] provide a closed form for the number of trees with n leaves and dimension less than h, i.e. for $\boldsymbol{\nu}^{(h)}(a^{n-1}c^n)$. They show that the expected dimension of a random binary tree with n leaves is tightly concentrated around $1/2 \log_2 n$. This implies a much faster convergence of Newton's method in the case of G_L. We conjecture that a similar result can also be derived for arbitrary context-free grammars.

In the idempotent case, we can use a result of [21] to obtain for a given context-free grammar G a Presburger formula of size linear in $|G|$ defining its Parikh image. It would be interesting, if one could generalize this procedure to semirings collapsed at k as the result of Sec. 5.1 in general leads to very large expressions.

References

1. Bloom, S.L., Ésik, Z.: Axiomatizing rational power series over natural numbers. Inf. Comput. 207(7), 793–811 (2009)
2. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. Int. J. Found. Comput. Sci. 14(4), 551 (2003)
3. Bozapalidis, S.: Equational elements in additive algebras. Theory Comput. Syst. 32(1), 1–33 (1999)
4. Esparza, J., Ganty, P., Kiefer, S., Luttenberger, M.: Parikh's theorem: A simple and direct automaton construction. Inf. Process. Lett. 111(12), 614–619 (2011)
5. Esparza, J., Kiefer, S., Luttenberger, M.: An Extension of Newton's Method to ω-Continuous Semirings. In: Harju, T., Karhumäki, J., Lepistö, A. (eds.) DLT 2007. LNCS, vol. 4588, pp. 157–168. Springer, Heidelberg (2007)
6. Esparza, J., Kiefer, S., Luttenberger, M.: On Fixed Point Equations over Commutative Semirings. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 296–307. Springer, Heidelberg (2007)
7. Esparza, J., Kiefer, S., Luttenberger, M.: Newtonian program analysis. J. ACM 57(6), 33 (2010)
8. Etessami, K., Yannakakis, M.: Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. J. ACM 56(1) (2009)
9. Flajolet, P., Raoult, J.C., Vuillemin, J.: The number of registers required for evaluating arithmetic expressions. Theor. Comput. Sci. 9, 99–125 (1979)
10. Foster, J.N., Karvounarakis, G.: Provenance and data synchronization. IEEE Data Eng. Bull. 30(4), 13–21 (2007)
11. Ganty, P., Majumdar, R., Monmege, B.: Bounded Underapproximations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 600–614. Springer, Heidelberg (2010)
12. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS, pp. 31–40 (2007)
13. Luttenberger, M., Schlund, M.: An extension of Parikh's theorem beyond idempotence. Tech. rep., TU München (2011), `http://arxiv.org/abs/1112.2864`
14. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. J. Autom. Lang. Comb. 7(3), 321–350 (2002)
15. Petre, I.: Parikh's theorem does not hold for multiplicities. J. Autom. Lang. Comb. 4(1), 17–30 (1999)
16. Pilling, D.L.: Commutative regular equations and Parikh's theorem. J. London Math. Soc., 663–666 (1973)
17. Pivoteau, C., Salvy, B., Soria, M.: Algorithms for combinatorial structures: Well-founded systems and newton iterations. J. Comb. Theory, Ser. A 119(8), 1711–1773 (2012)
18. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
19. Rozenberg, G.: Handbook of formal languages: Word, language, grammar, vol. 1. Springer (1997)
20. Schwoon, S., Jha, S., Reps, T.W., Stubblebine, S.G.: On generalized authorization problems. In: CSFW, pp. 202–218 (2003)
21. Verma, K.N., Seidl, H., Schwentick, T.: On the Complexity of Equational Horn Clauses. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 337–352. Springer, Heidelberg (2005)

Counting Minimal Symmetric Difference NFAs

Brink van der Merwe[1], Mark Farag[2], and Jaco Geldenhuys[1]

[1] Department of Computer Science
Stellenbosch University, Private Bag X1, 7602 Matieland, South Africa
{abvdm,jaco}@cs.sun.ac.za
[2] Department of Mathematics
Fairleigh Dickinson University, 1000 River Rd., Teaneck, NJ 07666 USA
mfarag@fdu.edu

Abstract. A result of Nicaud states that the number of distinct unary regular string languages recognized by minimal deterministic finite automata (DFAs) with n states is asymptotically equal to $n2^{n-1}$. We consider the analogous question for symmetric difference automata ($\mathbb{Z}_2$-NFAs), and show that precisely 2^{2n-1} unary languages are recognized by n-state minimal $\mathbb{Z}_2$-NFAs.

Keywords: weighted automata, succinctness, non-determinism.

1 Introduction

In contrast to (usual) non-deterministic finite automata (NFAs), which accept a string if there is at least one path from an initial state of a NFA to a final state, symmetric difference non-deterministic finite automata ($\mathbb{Z}_2$-NFAs) accept strings which have an odd number of accepting paths. In more abstract terms, $\mathbb{Z}_2$-NFAs are obtained when working over the semiring $\mathbb{Z}_2$ (i.e. the integers modulo 2 with the usual addition and multiplication), instead of the Boolean semiring, as in the case of (usual) NFAs. Alternatively, one may define $\mathbb{Z}_2$-NFAs just as one defines NFAs, but rather use the symmetric difference set operation (instead of set union), when determinizing. The language accepted by a $\mathbb{Z}_2$-NFA is then defined to be the language accepted by the deterministic finite automaton (DFA) obtained when determinizing the $\mathbb{Z}_2$-NFA under consideration.

In this paper, minimal $\mathbb{Z}_2$-NFAs refer to $\mathbb{Z}_2$-NFAs which are minimal in terms of their number of states. When counting minimal DFAs with n states, one usually considers only one of the possible n factorial DFAs that are obtained by relabeling the states of a DFA. In the case of $\mathbb{Z}_2$-NFAs, we replace the concept of relabeling of states by the more general notion of making a change of basis on a $\mathbb{Z}_2$-NFA. Two $\mathbb{Z}_2$-NFAs related by a change of basis accept the same language. It is also the case that if two minimal $\mathbb{Z}_2$-NFAs accept the same language, then they are related by a change of basis. Also, two $\mathbb{Z}_2$-NFAs related by a change of basis are either both or neither minimal. When we refer to counting minimal n-state $\mathbb{Z}_2$-NFAs, we mean that we are counting equivalence classes of minimal $\mathbb{Z}_2$-NFAs, where two $\mathbb{Z}_2$-NFAs are equivalent if they are related by a change of basis. This is thus equivalent to counting the number of languages recognized by minimal $\mathbb{Z}_2$-NFAs with n states.

In contrast to (usual) NFAs, for which minimization is PSPACE complete, it is shown in [3], [13] and in the chapter, Rational and Recognizable Power Series, in [5], that $\mathbb{Z}_2$-NFAs can be minimized in cubic time in the number of states of the $\mathbb{Z}_2$-NFA being minimized, or more precisely, in time proportional to $O(|\Sigma|n^3)$, where Σ is the input alphabet and n is the number of states in the $\mathbb{Z}_2$-NFA being minimized.

In [8], it is shown that minimal $\mathbb{Z}_2$-NFAs are closely linked to minimal DFAs, since if a minimal $\mathbb{Z}_2$-NFA is determinized, a minimal DFA is obtained. Minimal $\mathbb{Z}_2$-NFAs can thus be considered as compact representations of minimal DFAs.

In [2] it was shown that, similar to (minimal) DFAs, (minimal) $\mathbb{Z}_2$-NFAs can be learned in polynomial time in the setting of Angluin learning. This is most likely not the case for NFAs. Thus from the perspective of Angluin learning, it is interesting to have an indication of how much more succinctly languages can be represented with $\mathbb{Z}_2$-NFAs, compared to DFAs.

We show that the number of distinct unary regular string languages recognized by n-state minimal $\mathbb{Z}_2$-NFAs is precisely equal to 2^{2n-1}. Thus n-state minimal $\mathbb{Z}_2$-NFAs recognize asymptotically a factor of $2^n/n$ more languages than n-state minimal DFAs. We also find that the number of nonempty unary regular languages recognized by n-state (not necessarily minimal) $\mathbb{Z}_2$-NFAs equals $\frac{1}{3}(2^{2n+1}-2)$, which also represents an increase by an exponential factor over the corresponding number in the DFA case.

The layout of this paper is as follows. In the next section we define $\mathbb{Z}_2$-NFAs and related concepts, and also provide known results for minimal $\mathbb{Z}_2$-NFAs required in the remainder of the paper. Next, we give two normal forms for linearly accessible (and, therefore, minimal) $\mathbb{Z}_2$-NFAs. This is followed by a section with our results on counting unary $\mathbb{Z}_2$-NFAs. Finally we give our conclusions and suggest avenues for future work.

2 Definitions and Basic Results

One may regard $\mathbb{Z}_2$-NFAs as weighted automata over the semiring $\mathbb{Z}_2$, but in order to stay consistent with previous literature on the topic, we define $\mathbb{Z}_2$-NFAs as NFAs for which the symmetric difference set operation is used instead of union for determinization.

Definition 1. *([12]) A $\mathbb{Z}_2$-NFA $\mathcal{N}$ is a tuple $(Q, \Sigma, \delta, I, F)$, where Q is a finite nonempty set of states, Σ is a finite nonempty input alphabet, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ a set of final states, and $\delta : Q \times \Sigma \to 2^Q$ a transition function.*

The transition function δ is extended to $\delta : 2^Q \times \Sigma \to 2^Q$, by defining $\delta(P, a) = \bigoplus_{q \in P} \delta(q, a)$, for any $a \in \Sigma$ and $P \in 2^Q$, where $\oplus$ denotes the symmetric difference set operation. We define $\delta^* : 2^Q \times \Sigma^* \to 2^Q$ by $\delta^*(P, \epsilon) = P$ and

$\delta^*(P, aw) = \delta^*(\delta(P, a), w)$ for any $a \in \Sigma$, $w \in \Sigma^*$ and $P \in 2^Q$. Through a standard abuse of notation, we denote δ^* by δ.

Let $\mathcal{N} = (Q, \Sigma, \delta, I, F)$ be a $\mathbb{Z}_2$-NFA and let w be a word in Σ^*. Then $\mathcal{N}$ *accepts* w if and only if $|F \cap \delta(I, w)| \bmod 2 \neq 0$, that is, if and only if there is an odd number of paths for w from an initial state to a final state in $\mathcal{N}$. Note that this definition of acceptance for $\mathbb{Z}_2$-NFAs is equivalent to the definition of acceptance for weighted automata, with weights from $\mathbb{Z}_2$. As usual, the *language recognized* by $\mathcal{N}$, denoted by $L(\mathcal{N})$, is the set of all words accepted by $\mathcal{N}$.

By determinizing $\mathcal{N}$, we get a complete DFA $\mathcal{N}^{\mathbb{D}}$, which is defined next.

Definition 2. *([8]) Let $\mathcal{N} = (Q, \Sigma, \delta, I, F)$ be a $\mathbb{Z}_2$-NFA. Then the complete DFA $\mathcal{N}^{\mathbb{D}} = (Q^D, \Sigma, \delta^D, q_0, F^D)$, obtained by determinizing $\mathcal{N}$, is defined as follows:*

- *$Q^D = \{\delta(I, w) \mid w \in \Sigma^*\}$;*
- *for $J \in Q^D \subseteq 2^Q$, and $a \in \Sigma$, $\delta^D(J, a) = \oplus_{q \in J} \delta(q, a)$, with $\delta^D(\emptyset, a) = \emptyset$ for all $a \in \Sigma$, if $\emptyset \in Q^D$;*
- *the start state q_0 of $\mathcal{N}^{\mathbb{D}}$ is the set I;*
- *the set of final states F^D of $\mathcal{N}^{\mathbb{D}}$ is $\{K \in Q^D \mid |K \cap F| \bmod 2 \neq 0\}$.*

It follows directly from the definition of the function $\delta : 2^Q \times \Sigma^* \to 2^Q$, that $\mathcal{N}^{\mathbb{D}}$ is equivalent to $\mathcal{N}$. From the equivalence of $\mathcal{N}^{\mathbb{D}}$ and $\mathcal{N}$, and from the fact that the language recognized by a DFA do not change if we interpret the DFA as a $\mathbb{Z}_2$-NFA, we have that the class of $\mathbb{Z}_2$-NFAs recognizes the class of regular languages.

Example 3. In this example we consider a $\mathbb{Z}_2$-NFA $\mathcal{N}$, given in Figure 1, for the regular expression $(a+b)^*b(a+b)$. By using the determinization procedure given above, we obtain $\mathcal{N}^{\mathbb{D}}$. If we change the $\mathbb{Z}_2$-NFA in Figure 1(a), so that q_2 is also an accept state, then the state labeled by $\{q_1, q_2, q_3\}$ in Figure 1(b) will no longer be an accept state. In Figure 2 an unambiguous NFA, i.e. an NFA where each string has at most one accepting path, is given for the language $(a+b)^*b(a+b)$. Note that we may interpret an unambiguous NFA as a $\mathbb{Z}_2$-NFA without changing the language being recognized. It is straightforward to generalize this example to the class of languages $(a+b)^*b(a+b)^k$, where each positive integer $k \geq 1$ gives another language in the class of languages under consideration. For this class of examples, the minimal DFA for each language has exponentially more states than a corresponding minimal $\mathbb{Z}_2$-NFA (or unambiguous NFA).

We encode the transition function of a unary $\mathbb{Z}_2$-NFA $\mathcal{N}$ (with $\Sigma = \{a\}$), as a matrix $m(\delta(a))$ with entries from $\mathbb{Z}_2$, as follows:

$$m(\delta(a))_{ij} = \begin{cases} 1 \text{ if } q_j \in \delta(q_i, a) \\ 0 \text{ otherwise,} \end{cases}$$

and successive matrix multiplications over $\mathbb{Z}_2$ reflect the subset construction on $\mathcal{N}$.

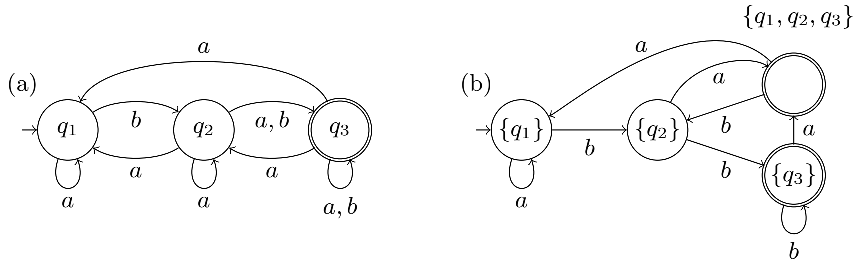


Fig. 1. (a) $\mathbb{Z}_2$-NFA $\mathcal{N}$ for the regular expression $(a+b)^*b(a+b)$, and (b) $\mathcal{N}^{\mathbb{D}}$, for Example 3

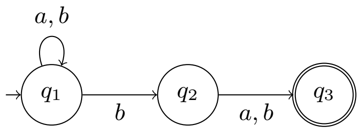


Fig. 2. an unambiguous NFA for $(a+b)^*b(a+b)$, for Example 3

We refer to $m(\delta(a))$ as the *transition matrix* of $\mathcal{N}$, and the *characteristic polynomial* of $\mathcal{N}$ is $c(\mathcal{N}) = \det(m(\delta(a)) - x\mathbb{I})$, where $\mathbb{I}$ is the identity matrix over $\mathbb{Z}_2$ of the appropriate size.

Assume Q has n-states. By letting $q_1, q_2, \ldots, q_n$ be an arbitrary but fixed ordering on the elements of Q, we encode $B \subseteq Q$ as an n-dimensional row vector $v(B) \in \mathbb{Z}_2^n$, by defining $v(B)_i = 1$ if $q_i \in B$, and $v(B)_i = 0$ otherwise.

For any integer $k \geq 0$, the matrix product $v(I)m(\delta(a))^k$ encodes the states reachable from the initial states after reading a^k. From the definition of acceptance and standard linear algebra, we have that $\mathcal{N}$ accepts a^k if and only if $v(I)m(\delta(a))^k v(F)^{\mathrm{T}} = 1$, where $v(F)^{\mathrm{T}}$ denotes the transpose of the row vector $v(F)$. In the case of non-unary $\mathbb{Z}_2$-NFAs, we associate a matrix $m(\delta(a))$ to each symbol $a \in \Sigma$. Then a word $w = a_1 \ldots a_k$, with $a_i \in \Sigma$, is accepted if and only if $v(I)m(\delta(a_1)) \ldots m(\delta(a_k))v(F)^{\mathrm{T}} = 1$.

The *range* of a $\mathbb{Z}_2$-NFA $\mathcal{N} = (Q, \Sigma, \delta, I, F)$ is defined as the linear subspace of 2^Q generated by subsets of the form $\delta(I, w)$. Recall that we denote by $\oplus$ the symmetric difference set operation.

Definition 4. *([8]) The* range $R(\mathcal{N})$ *of a* $\mathbb{Z}_2$*-NFA* $\mathcal{N} = (Q, \Sigma, \delta, I, F)$ *is the set of subsets of* Q *of the form* $\delta(I, w_1) \oplus \ldots \oplus \delta(I, w_k)$*, with* $w_1, \ldots, w_k \in \Sigma^*$, $k \geq 0$*, where we assume that for* $k = 0$ *we obtain the empty set.*

Assume that $|Q| = n$. Alternatively we can define $R(\mathcal{N})$ to be the linear subspace of the vector space $\mathbb{Z}_2^n$ generated by

$$\{v(I)\} \cup \{v(I)m(\delta(w_1)) \ldots m(\delta(w_k)) \mid k \geq 1 \text{ and } w_i \in \Sigma\}.$$

Definition 5. *([8],[13]) A* $\mathbb{Z}_2$*-NFA* $\mathcal{N}$ *is* linearly accessible *if* $R(\mathcal{N}) = 2^Q$.

If $|Q| = n$, then $\mathcal{N}$ is linearly accessible if

$$\{v(I)\} \cup \{v(I)m(\delta(w_1))\dots m(\delta(w_k)) \mid k \geq 1 \text{ and } w_i \in \Sigma\}$$

spans $\mathbb{Z}_2^n$. So if $\mathcal{N}$ is unary (with $\Sigma = \{a\}$), $\mathcal{N}$ is linearly accessible if (and only if) $\{v(I)m(\delta(a))^k \mid k = 0, 1, \dots, n-1\}$ spans $\mathbb{Z}_2^n$, or equivalently, if (and only if) $\{v(I)m(\delta(a))^k \mid k = 0, 1, \dots, n-1\}$ is linearly independent.

Definition 6. *The* mirror image *or* reverse of a string $w = a_1 \dots a_n$ *is the string* $w^R = a_n \dots a_1$. *The* reverse L^R of a language L *is defined to be* $\{w^R \mid w \in L\}$.

Definition 7. *([8]) Given a $\mathbb{Z}_2$-NFA, $\mathcal{N} = (Q, \Sigma, \delta, I, F)$, we define its* reverse *as $\mathcal{N}^{\mathbb{R}} = (Q, \Sigma, \delta^{\mathbb{R}}, F, I)$, where $q \in \delta^{\mathbb{R}}(p, a)$ if and only if $p \in \delta(q, a)$.*

In terms of vectors and matrices, the initial and final vectors are exchanged, and the transpose (that is, exchanging rows and columns) of the transition matrices of $\mathcal{N}$ is taken, in order to obtain the initial and final vectors and transition matrices of $\mathcal{N}^{\mathbb{R}}$. Note that

$$v(I)m(\delta(a_1))\dots m(\delta(a_k))v(F)^{\mathrm{T}} = v(F)m(\delta(a_k))^{\mathrm{T}}\dots m(\delta(a_1))^{\mathrm{T}}v(I)^{\mathrm{T}}.$$

Thus $L(\mathcal{N}^{\mathbb{R}}) = (L(\mathcal{N}))^R$. For unary $\mathbb{Z}_2$-NFAs, $L(\mathcal{N}^{\mathbb{R}}) = L(\mathcal{N})$.

Theorem 8. *([13]) A $\mathbb{Z}_2$-NFA $\mathcal{N}$ is minimal if and only if $\mathcal{N}$ and $\mathcal{N}^{\mathbb{R}}$ are linearly accessible.*

Definition 9. *([7]) Let L be a (regular) language. The* left quotient *of L by a word $w \in \Sigma^*$, is the language $w^{-1}L = \{x \in \Sigma^* \mid wx \in L\}$.*

If $\mathcal{D}$ is a complete minimal DFA recognizing L, then the right language of every state of $\mathcal{D}$ is a unique left quotient of L, and each left quotient is a right language of one of the states of $\mathcal{D}$. The *index* of a regular language L is the number of states in the complete minimal DFA recognizing L.

The *Hankel matrix* ([3]) for $L \subseteq \Sigma^*$, denoted by $H(L)$, is defined as follows. Assume that $w_1, w_2, \dots$ is a fixed ordering on Σ^*. Denote by $H(L)_{i,j} \in \mathbb{Z}_2$ the entry in column i and row j of the Hankel matrix. Then $H(L)_{i,j} := 1$ if $w_i w_j \in L$, and $H(L)_{i,j} = 0$ otherwise. Note that the Hankel matrix has infinitely many rows and columns. The *dimension* of a language L, denoted by $dim(L)$, is defined to be the rank of $H(L)$. The number of states in a minimal $\mathbb{Z}_2$-NFA recognizing L, with $L \neq \emptyset$, is equal to $dim(L)$. The Hankel matrix for the empty language is the matrix with 0's in all entries. So since this matrix has rank 0, we exclude the empty language from our discussions. From [13] we have the following relationship between $dim(L)$ and $index(L)$:

$$\log_2(index(L)) \leq dim(L) \leq index(L)\,.$$

Definition 10. *([13]) Let $\mathcal{N}$ be a n-state $\mathbb{Z}_2$-NFA with initial vector $v(I)$, final vector $v(F)$, and transition matrices $m(\delta(a))$, for all $a \in \Sigma$, and let A be an $n \times n$ non-singular matrix with inverse A^{-1}. Then we denote by $\mathcal{N}_A$ the $\mathbb{Z}_2$-NFA with initial vector $v(I)A$, final vector $(A^{-1}v(F)^{\mathrm{T}})^{\mathrm{T}}$, and transition matrices $A^{-1}m(\delta(a))A$, for $a \in \Sigma$. $\mathcal{N}_A$ is known as the $\mathbb{Z}_2$-NFA obtained from $\mathcal{N}$ by making a* change of basis *with the non-singular matrix A.*

The $\mathbb{Z}_2$-NFAs $\mathcal{N}$ and $\mathcal{N}_A$ are equivalent since:

$$\begin{aligned} & v(I)m(\delta(a_1))\dots m(\delta(a_k))v(F)^{\mathrm{T}} \\ = & v(I)AA^{-1}m(\delta(a_1))A\dots A^{-1}m(\delta(a_k))AA^{-1}v(F)^{\mathrm{T}}. \end{aligned}$$

It is shown in [13] that if $\mathcal{N}$ and $\mathcal{N}'$ are minimal $\mathbb{Z}_2$-NFAs for the same language L, then we can find a non-singular matrix A such that $\mathcal{N}' = \mathcal{N}_A$. It is important to note that the properties of being linearly accessible and being minimal are both preserved when making a change of basis.

3 Normal Forms for Unary $\mathbb{Z}_2$-NFAs

In this section, we give two normal forms for linearly accessible unary $\mathbb{Z}_2$-NFAs. Any linearly accessible unary $\mathbb{Z}_2$-NFA can be changed into either of these two normal forms by making an appropriate change of basis.

The *companion matrix* of the polynomial $f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + x^n \in \mathbb{Z}_2[x]$, is defined as

$$C(f) := \begin{bmatrix} 0 & 1 & 0 & 0\dots & 0 \\ 0 & 0 & 1 & 0\dots & 0 \\ \dots & & & & \\ 0 & 0 & 0 & 0\dots & 1 \\ a_0 & a_1 & \dots & & a_{n-1} \end{bmatrix}.$$

(Note for $n = 1$, this matrix is of the form $[a_0]$.)

We summarize some commonly known properties of such matrices as they will be used in the sequel. For proofs we refer the reader to [6], in which the transpose of our companion matrix form is used.

Theorem 11.

(a) The characteristic polynomial of the companion matrix $C(f)$ is f, and this is also the minimal polynomial of $C(f)$.

(b) An $n \times n$ matrix B is similar to the companion matrix of its characteristic polynomial, i.e. $C^{-1}BC$, for some non-singular matrix C, is equal to the companion matrix of the characteristic polynomial of B, if and only if the characteristic polynomial of B equals the minimal polynomial of B.

(c) An $n \times n$ matrix B is similar to the companion matrix of its characteristic polynomial if and only if $\{v, vB, vB^2, \dots, vB^{n-1}\}$ is linearly independent for some vector v.

In the first normal form, in Theorem 12 below, the initial state vector $v(I)$ is of the form $\mathbf{e}_1 := [1, 0, 0, \dots, 0]$, and the transition matrix $m(\delta(a))$ equals $C(c(\mathcal{N}))$, where $c(\mathcal{N})$ is the characteristic polynomial of the transition matrix of $\mathcal{N}$. Writing $c(\mathcal{N})$ as a product of powers of distinct irreducible factors, $p_1^{r_1}p_2^{r_2}\dots p_k^{r_k}$, then Theorem 13 below gives a second normal form for a linearly accessible $\mathbb{Z}_2$-NFA $\mathcal{N}$ as the disjoint union of k $\mathbb{Z}_2$-NFAs $\mathcal{N}_i$, for $i = 1, \dots, k$, where each $\mathcal{N}_i$ has an initial state vector of the form $[1, 0, 0, \ldots, 0]$, and transition matrix equal to $C(p_i^{r_i})$. It should be noted that the argument used in the proof of Theorem 12 below follows, among other places, from the algorithm given in [13] to obtain an equivalent linearly accessible weighted automata (with weights from a field) from any given weighted automaton. Although it is possible to extend the first normal to the non-unary case, we will not present this more general case since our main focus in the remainder of this paper is on unary $\mathbb{Z}_2$-NFAs.

Theorem 12. *Any linearly accessible unary $\mathbb{Z}_2$-NFA $\mathcal{N}$, is equivalent, under a change of basis, to a $\mathbb{Z}_2$-NFA with initial state vector $v(I) = \mathbf{e}_1$ and transition matrix $C(c(\mathcal{N}))$, where $c(\mathcal{N})$ is the characteristic polynomial of the transition matrix of $\mathcal{N}$.*

Proof. Given a linearly accessible n-state $\mathbb{Z}_2$-NFA $\mathcal{N}$, we let A be a non-singular matrix such that $v(I)A = \mathbf{e}_1$. Then $\mathcal{N}_A$, i.e. the $\mathbb{Z}_2$-NFA obtained by making a change of basis using A, has initial state vector $\mathbf{e}_1$. Now consider the first row $[t_{11}t_{12} \ldots t_{1n}]$ of the transition matrix of $\mathcal{N}_A$. Since this $\mathbb{Z}_2$-NFA is linearly accessible, there must exist some largest index $j_1 > 1$ for which $t_{1j_1} = 1$. If $j_1 = 2$ then do nothing; otherwise, for $k = 1, \ldots, j_1 - 1$, apply to the transition matrix in succession the elementary column operation that replaces column k (which we denote by C_k) with $C_k + t_{1k} \cdot C_{j_1}$. Now interchange C_{j_1} with C_2 to obtain a first row equal to $\mathbf{e}_2$. Observe that pre-multiplying the resulting transition matrix by the inverse of the product of matrices resulting from these operations does not affect row 1 since: a) pre-multiplying by the inverse of replacing C_k with $C_k + t_{1k} \cdot C_{j_1}$ is equivalent to replacing R_{j_1} with $R_{j_1} + t_{1k} \cdot R_k$, and b) pre-multiplying by the inverse of interchanging C_{j_1} with C_2 is equivalent to interchanging row j_1, R_{j_1}, with R_2. If $n = 2$, we are done; otherwise consider in succession, for $i = 2 \ldots n - 1$, row i, given by $[t_{i1}t_{i2} \ldots t_{in}]$, of the resulting transition matrix. By the linearly accessibility of the $\mathbb{Z}_2$-NFA, there must exist some largest index $j_i > i$ for which $t_{ij_i} = 1$. If $j_i = i + 1$ then do nothing; otherwise, for $k = 1 \ldots j_i - 1$, apply to the transition matrix in succession the elementary column operation that replaces column k with $C_k + t_{ik} \cdot C_{j_i}$. Now interchange C_{j_i} with C_{i+1} to obtain a transition matrix with row i equal to $\mathbf{e}_{i+1}$. Observe that pre-multiplying the resulting transition matrix by the inverse of the product of matrices resulting from these operations does not affect row i, and that none of the elementary column operations used affects $\mathbf{e}_1$, so that the result follows. $\square$

Since by making a change of basis on a $\mathbb{Z}_2$-NFA, we do not change the property of being (or not being) linearly accessible, and since $\mathbb{Z}_2$-NFAs of the form specified in the theorem above are linearly accessible, we have that a $\mathbb{Z}_2$-NFA $\mathcal{N}$ is linearly accessible if and only $\mathcal{N}_A$, for some non-singular matrix A, is of the form specified in the theorem above.

Note that the fact that for a linearly accessible unary $\mathbb{Z}_2$-NFA $\mathcal{N}$, we can find a non-singular matrix A such that the transition matrix of $\mathcal{N}_A$ is a companion matrix, follows directly from the definition of being linearly accessible and from (c) in Theorem 11 above, but to ensure that both the initial state vector of $\mathcal{N}_A$

is equal to $\mathbf{e}_1$ and the transition matrix of $\mathcal{N}_A$ is a companion matrix, requires more of an argument.

Theorem 13. *Any linearly accessible n-state unary $\mathbb{Z}_2$-NFA, is equivalent, under a change of basis, to a disjoint union of $\mathbb{Z}_2$-NFAs, each having an initial state vector of the form $[1, 0, 0, \ldots, 0]$, and a transition matrix which is a companion matrix of a polynomial that is a power of a different irreducible polynomial.*

Proof. For square matrices $A_1, A_2, \ldots, A_k$, each of size $n_l \times n_l$, for $l = 1, 2, \ldots, k$, we denote by $A_1 \oplus A_2 \oplus \ldots \oplus A_k$ the $n \times n$ matrix ($n = n_1 + n_2 + \ldots + n_k$), with the matrices $A_1, A_2, \ldots, A_k$ as blocks on the diagonal of A, and all other entries of A equal to 0. Let $\mathcal{N}$ be a linearly accessible $\mathbb{Z}_2$-NFA, and assume that $c(\mathcal{N})$, the characteristic polynomial of $\mathcal{N}$, factorizes into distinct irreducible polynomials p_i with exponents r_i as follows: $p_1^{r_1} p_2^{r_2} \ldots p_k^{r_k}$. Then from the Chinese remainder theorem for companion matrices (Theorem 5.31 in [10]), it follows that $B^{-1}C(f)B = C(p_1^{r_1}) \oplus C(p_2^{r_2}) \oplus \ldots \oplus C(p_k^{r_k})$, for some non-singular matrix B. Thus $\mathcal{N}_B$ has transition matrix $C(p_1^{r_1}) \oplus C(p_2^{r_2}) \oplus \ldots \oplus C(p_k^{r_k})$, and we may regard $\mathcal{N}_B$ as the disjoint union of k $\mathbb{Z}_2$-NFAs, each having a characteristic polynomial which is the power of a different irreducible polynomial in $\mathbb{Z}_2[x]$. Each of these k $\mathbb{Z}_2$-NFAs must be linearly accessible, otherwise $\mathcal{N}$ would not be linearly accessible. Theorem 12 can now be used to put each of these k $\mathbb{Z}_2$-NFAs in the normal form specified in Theorem 12. □

Example 14. In this example we list all minimal (non-equivalent) unary $\mathbb{Z}_2$-NFAs (over the alphabet $\{a\}$), in the normal forms given above, with one and with two states. For this example, it is only the case where $c(\mathcal{N}) = x(x + 1)$, in which the normal form of Theorem 13 leads to more than one $\mathbb{Z}_2$-NFA in the disjoint union, and where we thus have a difference between the two normal forms.

Minimal $\mathbb{Z}_2$-NFAs, in normal form, with one state:

Case 1: A single state, which is both an initial state and final state with no transitions. This $\mathbb{Z}_2$-NFA recognizes $\{\varepsilon\}$.

Case 2: A single state, which is both an initial state and final state, with a transition from the state to itself. This $\mathbb{Z}_2$-NFA recognizes Σ^*.

Next we consider 2-state minimal $\mathbb{Z}_2$-NFAs. To verify that these $\mathbb{Z}_2$-NFAs are minimal, one can either verify that their Hankel matrices have rank 2, or one can check that they do not recognize the empty language or any of the two languages listed for the 1-state minimal $\mathbb{Z}_2$-NFAs above. Recall that for $\mathbb{Z}_2$-NFAs in the normal form of Theorem 12, we assume that only state 1 is initial, and that the transition matrices are of the form

$$\begin{bmatrix} 0 & 1 \\ a_0 & a_1 \end{bmatrix}.$$

So for each possible choice of a_0, a_1, we have to check which of the 4 possible choices of final states will lead to a minimal $\mathbb{Z}_2$-NFA. Since we have to pick at least some final states (otherwise the empty language will be recognized), we have in fact only three possible choices to consider.

Let $c(\mathcal{N}) = 1 + x + x^2$, then we have 3 minimal $\mathbb{Z}_2$-NFAs. Number the states as 1 and 2. With this choice of $c(\mathcal{N})$, $a_0 = 1, a_1 = 1$, and the $\mathbb{Z}_2$-NFAs under consideration will have 1 as initial state, and transitions $1 \to 2, 2 \to 1, 2 \to 2$. If only 1 is final, the language $\{\epsilon, a^2, a^3, a^5, a^6, a^8, \ldots\}$ is recognized. If only 2 is final, the language $\{a, a^2, a^4, a^5, a^7, a^8, \ldots\}$ is recognized. If 1 and 2 are final, the language $\{\epsilon, a, a^3, a^4, a^6, a^7, \ldots\}$ is recognized.

Now let $c(\mathcal{N}) = 1 + x^2 = (1 + x)^2$, then we have 2 minimal $\mathbb{Z}_2$-NFAs. Again 1 is initial, and given the choice of $c(\mathcal{N})$, we have transitions $1 \to 2, 2 \to 1$. If only 1 is final, the language $\{\epsilon, a^2, a^4, a^6, \ldots\}$ is recognized. If only 2 is final, the language $\{a, a^3, a^5, \ldots\}$ is recognized. Note that if we make both 1 and 2 final, Σ^* is recognized, but this language can also be recognized by a 1-state $\mathbb{Z}_2$-NFA, so this 2-state $\mathbb{Z}_2$-NFA, with both states final, is not minimal.

Now let $c(\mathcal{N}) = x + x^2 = x(1 + x)$, then we have only one minimal automaton. Again 1 is initial. Given $c(\mathcal{N})$, we have transitions $1 \to 2, 2 \to 2$. It is only the case where 2 is final (and 1 is not final), that we obtain a minimal $\mathbb{Z}_2$-NFA. This minimal $\mathbb{Z}_2$-NFA recognizes $\{a, a^2, a^3, a^4, \ldots\}$. Note that the disjoint union of the two 1-state minimal $\mathbb{Z}_2$-NFAs also recognizes this language.

Finally, let $c(\mathcal{N}) = x^2$. In this case we have two minimal $\mathbb{Z}_2$-NFAs. Again 1 is initial. Given $c(\mathcal{N})$, we have a transition $1 \to 2$. If 2 is final, the language $\{a\}$ is recognized. If 1 and 2 are final, the language $\{\epsilon, a\}$ is recognized. If only 1 is final, the language $\{\epsilon\}$ is recognized, but this language can also be recognized by a 1-state $\mathbb{Z}_2$-NFA.

4 Results on Counting Unary NFAs

Denote by $F_{\ell,\mathbb{Z}_2}(n)$ the number of distinct languages on an alphabet of size ℓ recognized by minimal $\mathbb{Z}_2$-NFAs with n states. Similarly, denote by $G_{\ell,\mathbb{Z}_2}(n)$ the number of distinct nonempty languages on an alphabet of size ℓ recognized by (not necessarily minimal) $\mathbb{Z}_2$-NFAs with n states. We show that $F_{1,\mathbb{Z}_2}(n) = 2^{2n-1}$ and that, consequently, $G_{1,\mathbb{Z}_2}(n) = \frac{1}{3}(2^{2n+1} - 2)$.

In this section, "normal form" refers to the normal form for linearly accessible unary $\mathbb{Z}_2$-NFAs given in Theorem 12.

The next result is used in Theorem 16 to show that if two minimal unary $\mathbb{Z}_2$-NFA have different normal forms, then they recognize different languages.

For a $\mathbb{Z}_2$-NFA $\mathcal{N} = (Q, \Sigma, \delta, I, F)$, we denote the $\mathbb{Z}_2$-NFA $(Q, \Sigma, \delta, I, F')$ by $\mathcal{N}_{F'}$

Proposition 15. *Let $\mathcal{N}$ be a linearly accessible $\mathbb{Z}_2$-NFA. Then $L(\mathcal{N}_{F_1}) \neq L(\mathcal{N}_{F_2})$ for $F_1 \neq F_2$.*

Proof. Since $\mathcal{N}$ is linearly accessible, there exists a word $w = w_1 \ldots w_k \in \Sigma^*$ such that $v(I)m(\delta(w_1)) \ldots m(\delta(w_k))v(F_1)^T \neq v(I)m(\delta(w_1)) \ldots m(\delta(w_k))v(F_2)^T$. □

Theorem 16. *Assume $\mathcal{N}$ and $\mathcal{N}'$ are different minimal unary $\mathbb{Z}_2$-NFAs in normal form. Then $L(\mathcal{N}) \neq L(\mathcal{N}')$.*

Proof. We give a proof by contradiction. Assume $\mathcal{N}$ and $\mathcal{N}'$ are in normal form with $L(\mathcal{N}) = L(\mathcal{N}')$. Since $\mathcal{N}$ and $\mathcal{N}'$ are minimal, and $L(\mathcal{N}) = L(\mathcal{N}')$, we have from [13] that $\mathcal{N}' = \mathcal{N}_A$, for some non-singular matrix A. But since the characteristic polynomial of a matrix does not change if we make a change of basis, we have that $c(\mathcal{N}) = c(\mathcal{N}')$, and thus from Theorem 11 (c), $\mathcal{N}$ and $\mathcal{N}'$ have equal transition matrices. Since Proposition 15 implies that $\mathcal{N}$ and $\mathcal{N}'$ must also have equal final states, the result follows. □

We now recall some concepts from ring and module theory that will be used in the next theorem. Let R be an associative ring. Then we denote the set of multiplicative units in R by $\mathcal{U}(R)$. In our case, R will be of the form $\mathbb{Z}_2[x]/\langle f\rangle$, the quotient ring of the polynomial ring $\mathbb{Z}_2[x]$ modulo the ideal generated by a polynomial $f \in \mathbb{Z}_2[x]$. Recall that $\langle f\rangle$ simply consists of all multiples of f by elements of $\mathbb{Z}_2[x]$ and that the elements of $\mathbb{Z}_2[x]/\langle f\rangle$ are all cosets of the form $g + \langle f\rangle$, where g can be taken to have degree less than the degree of f. The set $\mathbb{Z}_2[x]/\langle f\rangle$ has the structure of a ring when endowed, for $g_1, g_2 \in \mathbb{Z}_2[x]$, with the addition operation $(g_1+\langle f\rangle) + (g_2+\langle f\rangle) = (g_1+g_2)+\langle f\rangle$ and the multiplication operation $(g_1 + \langle f\rangle) * (g_2 + \langle f\rangle) = (g_1g_2) + \langle f\rangle$, where g_1g_2 may be replaced by the remainder of g_1g_2 upon division by f. Thus $g_1 + \langle f\rangle$ is a unit of $\mathbb{Z}_2[x]/\langle f\rangle$ if there exists an element $g_2 + \langle f\rangle$ for which the remainder of g_1g_2 upon division by f is 1.

Given an associative ring R with identity 1_R, an abelian group M is a right R-module if there is a (scalar) multiplication map $\mu : M \times R \to M$ given via $(m, r)\mu = mr$ and satisfying: for all $m, n \in M$ and all $r, s \in R$, 1) $(m + n)r = mr + nr$, 2) $m(r + s) = mr + ms$, 3) $m(rs) = (mr)s$, and 4) $m1_R = m$. Similar to the situation for rings, a quotient module M/S may be constructed for a submodule S of M by defining multiplication and addition for the set of all cosets of the form $m + S$, where $m \in M$, as follows: for $r \in R$, $m, n \in M$, $(m+S)\cdot r = mr+S$ and $(m+S)+(n+S) = (m+n)+S$. The first isomorphism theorem for modules states that, given a homomorphism $\phi : M \to N$ between right R-modules M and N, the image of ϕ is isomorphic to the quotient module $M/\ker(\phi)$, where $\ker(\phi)$ is the submodule of all elements of M sent to 0_N by ϕ. In particular, if ϕ is onto, N is isomorphic to $M/\ker(\phi)$. The reader may refer to [1] for more basic information about rings, quotients, modules, and isomorphism theorems.

In the next result we determine how many choices for final states in a unary linearly accessible $\mathbb{Z}_2$-NFA will lead to a minimal $\mathbb{Z}_2$-NFA.

Theorem 17. *Let $\mathcal{N} = (Q, \{a\}, \delta, I, F)$ be a linearly accessible unary $\mathbb{Z}_2$-NFA. Then $\{\mathcal{N}_{F'} \mid F' \subseteq Q$ and $\mathcal{N}_{F'}$ is minimal $\}$ and $\mathcal{U}(\mathbb{Z}_2[x]/\langle c(\mathcal{N})\rangle)$ have the same cardinality.*

Proof. Since $\mathcal{N}$ is linearly accessible, the minimal polynomial of $\mathcal{N}$ is also equal to $c(\mathcal{N})$. Also, from Theorem 8, $\mathcal{N}_{F'}$ is minimal if and only if $(\mathcal{N}_{F'})^{\mathbb{R}}$ is linearly accessible. Note that $(\mathcal{N}_{F'})^{\mathbb{R}}$ has initial states F' and transition matrix $m(\delta(a))^T$. We have that $(\mathcal{N}_{F'})^{\mathbb{R}}$ is linearly accessible if and only if the vectors

$$v(F'), v(F')m(\delta(a))^T, v(F')(m(\delta(a))^T)^2, \ldots, v(F')(m(\delta(a))^T)^{n-1}$$

spans $\mathbb{Z}_2^n$, or equivalently, if these vectors are linearly independent. Since a matrix and its transpose have the same characteristic and minimal polynomial, the characteristic and minimal polynomial of $m(\delta(a))^T$ are also equal to $c(\mathcal{N})$. Thus from Theorem 11 we can find at least one set $\hat{F} \subseteq Q$, such that

$$v(\hat{F}), v(\hat{F})m(\delta(a))^T, v(\hat{F})(m(\delta(a))^T)^2, \ldots, v(\hat{F})(m(\delta(a))^T)^{n-1}$$

spans $\mathbb{Z}_2^n$. Now consider $(\mathbb{Z}_2)^n$ as a right $\mathbb{Z}_2[x]$-module, where $v.r$ (v in $(\mathbb{Z}_2)^n$, r in $\mathbb{Z}_2[x]$) is defined such that $v.x = vm(\delta(a))^T$. Thus since $(\mathbb{Z}_2)^n$ is spanned by the n vectors $v(\hat{F}), v(\hat{F})m(\delta(a))^T, v(\hat{F})(m(\delta(a))^T)^2, \ldots, v(\hat{F})(m(\delta(a))^T)^{n-1}$, the modules $(\mathbb{Z}_2)^n$ and $\mathbb{Z}_2[x]/\langle c(\mathcal{N})\rangle$ are isomorphic as right $\mathbb{Z}_2[x]$-modules.

Note that $(\mathcal{N}_{F'})^{\mathbb{R}}$ is linearly accessible if and only if $v(F')$ is a generator for the $\mathbb{Z}_2[x]$-module $\mathbb{Z}_2^n$. Thus the number of possible choices of F', such that $(\mathcal{N}_{F'})^{\mathbb{R}}$ is linearly accessible, is equal to the number of units in the factor ring $\mathbb{Z}_2[x]/\langle c(\mathcal{N})\rangle$, since $g + \langle c(\mathcal{N})\rangle$ is a generator for the right $\mathbb{Z}_2[x]$-module $\mathbb{Z}_2[x]/\langle c(\mathcal{N})\rangle$ if and only if $g + \langle c(\mathcal{N})\rangle$ is a unit in the factor ring $\mathbb{Z}_2[x]/\langle c(\mathcal{N})\rangle$. □

Theorem 18. *For $n \geq 1$, $F_{1,\mathbb{Z}_2}(n) = 2^{2n-1}$. Thus $G_{1,\mathbb{Z}_2}(n) = \frac{1}{3}(2^{2n+1} - 2)$.*

Proof. By Theorem 16 we have that $F_{1,\mathbb{Z}_2}(n)$ equals the number of minimal unary n-state $\mathbb{Z}_2$-NFAs in normal form. Let $f \in \mathbb{Z}_2[x]$ be a polynomial of degree n. Then from Theorem 17, the number of minimal $\mathbb{Z}_2$-NFAs with transition matrix $C(f)$, and thus the number of minimal unary n-state $\mathbb{Z}_2$-NFAs in normal form with characteristic polynomial f, is equal to the cardinality of $U(\mathbb{Z}_2[x]/\langle f\rangle)$. Thus $F_{1,\mathbb{Z}_2}(n) = \sum_{\deg(f)=n} \mid \mathcal{U}(\mathbb{Z}_2[x]/\langle f\rangle) \mid$. For a polynomial f of degree n, define $\Phi(f)$ as the number of polynomials in $\mathbb{Z}_2[x]$ of degree less than n that are relatively prime to f. Then from the definition of being relatively prime, we have that $\Phi(f)$ equals $\mid \mathcal{U}(\mathbb{Z}_2[x]/\langle f\rangle) \mid$. Observing that all nonzero polynomials in $\mathbb{Z}_2[x]$ are monic and taking $q = 2$ in Proposition 2.7[1] of [11] then yields the result: $F_{1,\mathbb{Z}_2}(n) = \sum_{\deg(f)=n} \mid \mathcal{U}(\mathbb{Z}_2[x]/\langle f\rangle) \mid = \sum_{\deg(f)=n} \Phi(f) = 2^{2n-1}$. It follows that $G_{1,\mathbb{Z}_2}(n) = \displaystyle\sum_{j=1}^{n} F_{1,\mathbb{Z}_2}(j) = \frac{1}{3}(2^{2n+1} - 2)$. □

5 Conclusions and Future Work

Comparing our result with Nicaud's result in [9], which shows the number of distinct unary languages recognized by minimal DFAs with n states to be asymptotically equal to $n2^{n-1}$, we find that n-state minimal $\mathbb{Z}_2$-NFAs recognize asymptotically a factor of $2^n/n$ more unary languages than their n-state minimal DFA counterparts. Since it is also shown in [4] that the number of distinct unary languages recognized by n-state DFAs is asympotically equal to $n2^n$, n-state $\mathbb{Z}_2$-NFAs recognize an exponential factor more unary languages than their n-state DFA counterparts. Also, if we denote by $G_{\ell,\mathbb{B}}(n)$ the number of distinct nonempty languages on an alphabet of size ℓ recognized by (not necessarily

[1] We would like to thank Stephan Wagner for pointing out this result from [11].

minimal) NFAs (or, equivalently, weighted automata over the Boolean semiring) with n states, we have from [4] that $G_{1,\mathbb{B}}(1) = 2$, $G_{1,\mathbb{B}}(2) = 8$, $G_{1,\mathbb{B}}(3) = 28$, $G_{1,\mathbb{B}}(4) = 87$ and $G_{1,\mathbb{B}}(5) = 268$. It is interesting to compare these numbers to $G_{1,\mathbb{Z}_2}(1), \ldots, G_{1,\mathbb{Z}_2}(5)$, since $G_{1,\mathbb{Z}_2}(1) = G_{1,\mathbb{B}}(1)$, and $G_{1,\mathbb{Z}_2}(n) > G_{1,\mathbb{B}}(n)$ for $n = 2, \ldots, 5$, with the ratios $G_{1,\mathbb{Z}_2}(n)/G_{1,\mathbb{B}}(n)$ also increasing for $n = 1, \ldots, 5$.

We note that it is possible to extend our results to automata over any finite field, though it is not clear how interesting it is to consider weighted automata over other finite fields than $\mathbb{Z}_2$. Avenues for future investigations include extending our results to the non-unary case as well as to tree automata.

References

1. Artin, M.: Algebra, 1st edn. Prentice-Hall, Incorporated (1991)
2. Beimel, A., Bergadano, F., Bshouty, N.H., Kushilevitz, E., Varricchio, S.: Learning functions represented as multiplicity automata. J. ACM 47(3), 506–530 (2000), http://doi.acm.org/10.1145/337244.337257
3. Berstel Jr., J., Reutenauer, C.: Rational series and their languages. Springer-Verlag New York, Inc., New York (1988)
4. Domaratzki, M., Kisman, D., Shallit, J.: On the number of distinct languages accepted by finite automata with n states. J. Autom. Lang. Comb. 7(4), 469–486 (2002), http://dl.acm.org/citation.cfm?id=782466.782472
5. Droste, M., Rahonis, G.: Weighted Automata and Weighted Logics on Infinite Words. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 49–58. Springer, Heidelberg (2006)
6. Horn, R.A., Johnson, C.R.: Matrix analysis. Cambridge University Press (1990)
7. Linz, P.: An Introduction to Formal Languages and Automata, 5th edn. Jones and Bartlett Publishers, Inc., USA (2011)
8. van der Merwe, B., Tamm, H., van Zijl, L.: Minimal DFA for Symmetric Difference NFA. In: Kutrib, M., Moreira, N., Reis, R. (eds.) DCFS 2012. LNCS, vol. 7386, pp. 307–318. Springer, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-31623-4
9. Nicaud, C.: Average State Complexity of Operations on Unary Automata. In: Kutyłowski, M., Wierzbicki, T., Pacholski, L. (eds.) MFCS 1999. LNCS, vol. 1672, pp. 231–240. Springer, Heidelberg (1999), http://dl.acm.org/citation.cfm?id=645728.667710
10. Norman, C.: Finitely Generated Abelian Groups and Similarity of Matrices Over a Field. Springer Undergraduate Mathematics. Springer (2012)
11. Rosen, M.: Number Theory in Function Fields. Graduate Texts in Mathematics. Springer (2002)
12. Van Zijl, L.: Random Number Generation with ⊕-NFAs. In: Watson, B.W., Wood, D. (eds.) CIAA 2001. LNCS, vol. 2494, pp. 263–273. Springer, Heidelberg (2003), http://dl.acm.org/citation.cfm?id=647268.721718
13. Vuillemin, J., Gama, N.: Compact Normal Form for Regular Languages as Xor Automata. In: Maneth, S. (ed.) CIAA 2009. LNCS, vol. 5642, pp. 24–33. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02979-0

Interval Logics and ωB-Regular Languages

Angelo Montanari[1] and Pietro Sala[2]

[1] Department of Mathematics and Computer Science, University of Udine, Italy
angelo.montanari@uniud.it

[2] Department of Pharmacology, University of Verona, Italy
pietro.sala@univr.it

Abstract. In the recent years, interval temporal logics are emerging as a workable alternative to more standard point-based ones. In this paper, we establish an original connection between these logics and ωB-regular languages. First, we provide a logical characterization of regular (resp., ω-regular) languages in the interval logic $AB\bar{B}$ of Allen's relations *meets*, *begun by*, and *begins* over finite linear orders (resp., $\mathbb{N}$). Then, we lift such a correspondence to ωB-regular languages by substituting $AB\bar{B}\bar{A}$ for $AB\bar{B}$ ($AB\bar{B}\bar{A}$ is obtained from $AB\bar{B}$ by adding a modality for Allen's relation *met by*). In addition, we show that new classes of extended (ω-)regular languages can be naturally defined in $AB\bar{B}\bar{A}$.

1 Introduction

In this paper, we establish an original connection between interval temporal logics (ITLs) and various classes of regular languages of finite and infinite words. In particular, we show that ω-regular languages extended with boundedness can be defined in the interval logic $AB\bar{B}\bar{A}$. ITLs provide a general framework for representing and reasoning about time, where standard (point-based) temporal logics can be recovered as suitable syntactic fragments. ITLs are characterized by high expressiveness and high computational complexity. One of the first ITLs proposed in the literature is Moszkowski's Propositional ITL (PITL), which has been successfully applied to the specification and verification of hardware components [14]. The application of interval-based formalisms to temporal reasoning in AI was investigated in [1]. Allen's Interval Algebra allows one to reason about all possible temporal relations between two (non-point) intervals in a linear order. A systematic logical study of interval reasoning started with Halpern and Shoham's work on the logic HS featuring one modality for each Allen's relation [9]. While decidability is a common feature of point-based temporal logics, undecidability rules over ITLs. The first such undecidability results were obtained for PITL by Moszkowski in [14]. General undecidability results for HS are given in [9] and further sharpened in [10]. For a long time, these results have discouraged the search for practical applications and further theoretical investigation on ITLs. This bleak picture started lightening up in the last few years when various non-trivial decidable fragments of HS have been identified [5–7, 11–13].

In this paper, we explore the relationships between ITLs and regular languages of finite and infinite words. While the consensus on what features regular languages of finite words must exhibit is unanimous (it largely relies on Myhill-Nerode characterization of these languages), the notion of regular ω-languages is somehow controversial. Recent work by (among others) Bojańczyk and Colcombet shows that ω-languages can be extended in various meaningful ways, preserving their decidability and closure properties [2–4]. As an example, extended ω-languages make it possible to constrain the distance consecutive occurrences of a given symbol to be (un)bounded, a property of interest in the specification of reactive systems. In the following, we provide a temporal logic characterization of ω-regular languages with boundedness (ωB-regular languages) and of some variants of them. We first show that regular (resp., ω-regular) languages can be defined in the interval logic $AB\bar{B}$ of Allen's relations *meets*, *begun by*, and *begins* over finite linear orders (resp., $\mathbb{N}$). Then, we lift such a correspondence to ωB-regular languages by adding a modality $\langle \bar{A} \rangle$, corresponding to Allen's relation *met by*, to $AB\bar{B}$. The distinctive feature of the proposed solution is that extended ω-languages can be encoded in the resulting logic $AB\bar{B}\bar{A}$ without resorting to any counter, that is, checking the satisfaction of conditions like boundedness in $AB\bar{B}\bar{A}$ does not require the precision in length measurements given by counters.

The paper is organized as follows. Section 2 introduces syntax and semantics of $AB\bar{B}$ and $AB\bar{B}\bar{A}$, and we provide basic notation, tools, and results. Section 3 defines an encoding of finite and ω-regular languages into $AB\bar{B}$ over finite linear orders and $\mathbb{N}$, respectively. Section 4 extends the encoding to ωB-regular languages, and beyond, by replacing $AB\bar{B}$ by $AB\bar{B}\bar{A}$. Section 5 shows that a decidable fragment of $AB\bar{B}\bar{A}$ suffices for checking (non)emptiness.

2 The Interval Temporal Logics $AB\bar{B}$ and $AB\bar{B}\bar{A}$

In this section, we give syntax and semantics of $AB\bar{B}$ and $AB\bar{B}\bar{A}$, and we summarize known (un)decidability results for them. Then, we provide an alternative interpretation of them over labeled grid-like structures.

Syntax and Semantics. $AB\bar{B}$ features three modalities $\langle A \rangle$, $\langle B \rangle$, and $\langle \bar{B} \rangle$ corresponding to Allen's relations A (*meets*), B (*begun by*), and $\bar{B}$ (*begins*), respectively. Formally, given a set $\mathcal{P}rop$ of proposition letters, formulas of $AB\bar{B}$ are built up from $\mathcal{P}rop$ using the Boolean connectives $\neg$ and $\vee$ and the modalities $\langle A \rangle$, $\langle B \rangle$, $\langle \bar{B} \rangle$. As usual, we make use of shorthands like $\varphi_1 \wedge \varphi_2$ for $\neg(\neg\varphi_1 \vee \neg\varphi_2)$, $[A]\varphi$ for $\neg\langle A \rangle\neg\varphi$, $[B]\varphi$ for $\neg\langle B \rangle\neg\varphi$, $[\bar{B}]\varphi$ for $\neg\langle \bar{B} \rangle\neg\varphi$, $\top$ for $p \vee \neg p$, and $\perp$ for $p \wedge \neg p$, with $p \in \mathcal{P}rop$. We interpret formulas of $AB\bar{B}$ in interval temporal structures over $\mathbb{N}$ endowed with the relations A, B, and $\bar{B}$. We identify any given ordinal $N \leq \omega$ with the prefix of length N of $\mathbb{N}$ and we accordingly define $\mathbb{I}(N)$ as the set of all closed intervals $[i, j]$, with $i, j \in N$ and $i \leq j$. A special role will be played by point-intervals (intervals $[i, i]$, for each $i \in N$) and unit-intervals (intervals $[i, i+1]$), which are captured by the formulas $\pi = [B]\perp$ and $\mathit{unit} = [B][B]\perp$, respectively. For any pair $[i, j], [i', j'] \in \mathbb{I}(N)$, A, B, and $\bar{B}$ are defined as follows ($\bar{B}$ is the inverse relation of B): (i) $[i, j]\ A\ [i', j']$ iff $j = i'$, (ii) $[i, j]\ B\ [i', j']$ iff $i = i'$ and $j' < j$, and (iii) $[i, j]\ \bar{B}\ [i', j']$ iff $i = i'$ and $j < j'$. The (non-strict) semantics of $AB\bar{B}$ is given in terms of interval models $M = \langle \mathbb{I}(N), A, B, \bar{B}, V \rangle$, where $V : \mathbb{I}(N) \to \mathscr{P}(\mathcal{P}rop)$ is the valuation function that assigns to every $[i,j] \in \mathbb{I}(N)$, the set of proposition letters $V([i,j])$ that are true on it. The truth of an $AB\bar{B}$-formula over $[i,j]$ in M is defined as follows:

(i) $M,[i,j] \models p$ iff $p \in V([i,j])$, for all $p \in \mathcal{AP}$;
(ii) $M,[i,j] \models \neg\psi$ iff it is not the case that $M,[i,j] \models \psi$;
(iii) $M,[i,j] \models \varphi \vee \psi$ iff $M,[i,j] \models \varphi$ or $M,[i,j] \models \psi$;
(iv) $M,[i,j] \models \langle X \rangle \psi$ iff there exists an interval $[i',j']$ such that $[i,j]\ X\ [i',j']$, and $M,[i',j'] \models \psi$, for $X \in \{A, B, \bar{B}\}$.

Given $M = \langle \mathbb{I}(N), A, B, \bar{B}, V \rangle$ and φ, we say that M *satisfies* φ if there is $[i,j]$ in $\mathbb{I}(N)$ such that $M,[i,j] \models \varphi$. Wlog, we may assume $i = 0$ (initial satisfiability). We say that φ is *satisfiable* if there exists an interval model that satisfies it.

$AB\bar{B}$ can be viewed as the join of two simpler HS fragments, namely, $B\bar{B}$ and A. Decidability of $B\overline{B}$ over $\mathbb{N}$ (and over finite linear orders) can be easily shown by embedding it into the (point-based) temporal logic of linear time LTL$[F,P]$ [8]; decidability of A follows from the expressive completeness of $A\overline{A}$ with respect to the two-variable fragment of first-order logic for binary relational structures over $\mathbb{N}$ (and over finite linear orders) [6]. $AB\bar{B}$ retains the simplicity of its constituents, but it improves a lot on their expressive power. $AB\bar{B}$ is expressive enough to model inherently interval-based conditions like accomplishments, to encode the standard until operator of linear temporal logic LTL, and to constrain interval length [13]. Such an increase in expressiveness is achieved at the cost of an increase in complexity: $B\overline{B}$ and A are respectively NP-complete and NEXPTIME-complete, while $AB\bar{B}$ is EXPSPACE-complete [13].

To cope with more expressive finite and ω-regular languages, we replace $AB\bar{B}$ by $AB\bar{B}\bar{A}$, which features an additional modality $\langle \bar{A} \rangle$ corresponding to Allen's relation *met by*. For every pair $[i,j],[i',j'] \in \mathbb{I}(N)$, the relation $\bar{A}$ is defined as follows: $[i,j]\ \bar{A}\ [i',j']$ iff $j' = i$. The semantics is the same, but for the insertion of $\{\bar{A}\}$ in $\{A, B, \bar{B}\}$ in clause *(iv)*. The addition of $\langle \bar{A} \rangle$ to $AB\bar{B}$ drastically changes the characteristics of the logic [12]. Decidability is preserved if it is interpreted over finite linear orders, but the satisfiability problem is not primitive recursive anymore, and the addition of any other modality from HS repository (except for the modalities for Allen's relations *before* and *after*, which are definable in $A\overline{A}$) yields undecidability. It becomes undecidable over $\mathbb{N}$ (in fact, over any class of infinite Dedekind-complete linear orders).

Compass Structures. $AB\bar{B}\bar{A}$ (and its fragment $AB\bar{B}$) can be interpreted over grid-like structures (the so-called compass structures [15]) by exploiting the natural bijection between intervals $I = [x,y]$ and points $p = (x,y)$ of an $N \times N$ grid such that $x \leq y$ (octant). As an example, Figure 1 depicts four intervals $I_0, .., I_3, I_4$ such that $I_0\ A\ I_1$, $I_0\ \bar{A}\ I_2$, $I_0\ B\ I_3$, and $I_0\ \bar{B}\ I_4$, together with the corresponding points $p_0, .., p_3, p_4$ of a discrete grid (relations $A, \bar{A}, B$, and $\bar{B}$ between pairs of intervals are mapped to corresponding spatial relations between pairs of points; for the sake of readability, we name the latter as the former ones).

Let M be an interval model and φ be an $AB\bar{B}\bar{A}$ formula. We pair intervals in M with the sets of subformulas of φ they satisfy. Let the *closure* $\mathcal{C}l(\varphi)$ of

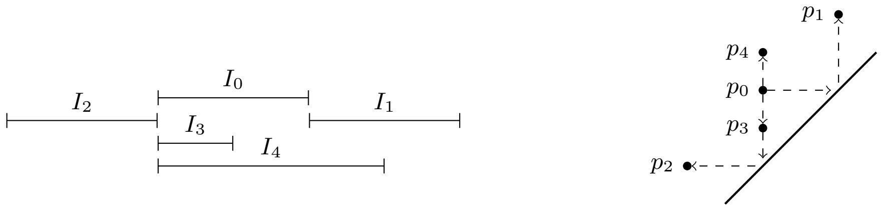


Fig. 1. A compass structure

φ be the set of all subformulas of φ and of their negations (we identify $\neg\neg\alpha$ with α, $\neg\langle A\rangle\alpha$ with $[A]\neg\alpha$, etc.), and let the *extended closure* $\mathcal{C}l^+(\varphi)$ be $\mathcal{C}l(\varphi)$ extended with all formulas of the forms $\langle R\rangle\alpha$ and $\neg\langle R\rangle\alpha$, with $R \in \{A, B, \bar{B}, \bar{A}\}$ and $\alpha \in \mathcal{C}l(\varphi)$. A *$\varphi$-atom* is a nonempty set $F \subseteq \mathcal{C}l^+(\varphi)$ such that (i) for each $\alpha \in \mathcal{C}l^+(\varphi)$, $\alpha \in F$ iff $\neg\alpha \notin F$ and (ii) for each $\gamma = \alpha \vee \beta \in \mathcal{C}l^+(\varphi)$, $\gamma \in F$ iff $\alpha \in F$ or $\beta \in F$. We denote by $\mathcal{A}_\varphi$ the set of all φ-atoms. The cardinalities of $\mathcal{C}l(\varphi)$ and $\mathcal{C}l^+(\varphi)$ are linear in the number $|\varphi|$ of subformulas of φ, while that of $\mathcal{A}_\varphi$ is at most exponential in $|\varphi|$. Let $F \in \mathcal{A}_\varphi$. For each $R \in \{A, B, \bar{B}, \bar{A}\}$, we denote by $\mathcal{R}eq_R(F)$ the set of all formulas $\alpha \in \mathcal{C}l^+(\varphi)$ such that $\langle R\rangle\alpha \in F$; moreover, we denote by $\mathcal{O}bs(F)$ the set of all formulas $\alpha \in \mathcal{C}l^+(\varphi)$ such that $\alpha \in F$. Making use of these sets, we define two basic relations between pairs of atoms F and G: (i) $F \xrightarrow{A} G$ iff $\mathcal{R}eq_A(F) = \mathcal{O}bs(G) \cup \mathcal{R}eq_B(G) \cup \mathcal{R}eq_{\bar{B}}(G)$ and $\mathcal{O}bs(F) \subseteq \mathcal{R}eq_{\bar{A}}(G)$, and (ii) $F \xrightarrow{B} G$ iff $\mathcal{O}bs(F) \cup \mathcal{R}eq_{\bar{B}}(F) \subseteq \mathcal{R}eq_{\bar{B}}(G)$, $\mathcal{R}eq_{\bar{B}}(G) \subseteq \mathcal{O}bs(F) \cup \mathcal{R}eq_{\bar{B}}(F) \cup \mathcal{R}eq_B(F)$, $\mathcal{O}bs(G) \cup \mathcal{R}eq_B(G) \subseteq \mathcal{R}eq_B(F)$, and $\mathcal{R}eq_B(F) \subseteq \mathcal{O}bs(G) \cup \mathcal{R}eq_B(G) \cup \mathcal{R}eq_{\bar{B}}(G)$.

Definition 1. *Let φ be an $AB\bar{B}\bar{A}$ formula. A (consistent and fulfilling)* labeled compass *(φ-)*structure *of length $N \leq \omega$ is a tuple $\mathcal{G} = (\mathbb{P}(N), A, B, \bar{B}, \bar{A}, \mathcal{L})$, where $\mathbb{P}(N) = \{(x, y) : 0 \leq x \leq y < N\}$ and $\mathcal{L} : \mathbb{P}(N) \mapsto \mathcal{A}_\varphi$ such that (i) for every $p, q \in \mathbb{P}(N)$ and $R \in \{A, B\}$, if $p\ R\ q$, then $\mathcal{L}(p) \xrightarrow{R} \mathcal{L}(q)$ (*consistency*), and (ii) for every $p \in \mathbb{P}(N)$, $R \in \{A, \bar{A}, B, \bar{B}\}$, and $\varphi \in \mathcal{R}eq_R(\mathcal{L}(p))$, there is $q \in \mathbb{P}(N)$ such that $p\ R\ q$ and $\varphi \in \mathcal{O}bs(\mathcal{L}(q))$ (*fulfillment*).*

We say that a labeled compass structure $\mathcal{G}$ *features* a formula φ if there is $p \in \mathbb{P}(N)$ such that $\varphi \in \mathcal{L}(p)$. The following theorem holds.

Theorem 2. *An $AB\bar{B}\bar{A}$ formula φ is satisfiable iff it is featured by some labeled compass structure.*

3 Encoding Finite and ω-Regular Languages in $AB\bar{B}$

Given a finite alphabet Σ, regular expressions R over Σ are defined as follows: $R ::= a \mid R+R \mid R \cdot R \mid R^*$, with $a \in \Sigma$. Each regular language can be captured by a regular expression, and vice versa. Similarly, ω-regular expressions O over Σ are defined as follows: $O ::= O+O \mid R \cdot O \mid R^\omega$, where R is a regular expression. Each ω-regular language can be expressed by an ω-regular expression, and vice versa. To determine an $AB\bar{B}$ counterpart ψ_R (resp., ψ_O) of a regular (resp., ω-regular) expression R (resp., O), we add a proposition letter a to $\mathcal{P}rop$ for each

$a \in \Sigma$ (hereafter, we assume the empty word ϵ to be included in Σ). Then, we force proposition letters in Σ to hold over unit-intervals only, and we constrain each unit interval to satisfy one and only one proposition letter in Σ. These conditions can be imposed by means of the following $AB\bar{B}$ formula (where $[G]\varphi$ is a shorthand for $[B]\varphi \wedge \varphi \wedge [\bar{B}]\varphi \wedge [B][A]\varphi \wedge [A][A]\varphi$–universal modality):

$$\psi_\Sigma = [G]((\text{ unit } \leftrightarrow \bigvee_{a\in\Sigma} a) \wedge \bigwedge_{a\in\Sigma} (a \rightarrow \bigwedge_{a'\in\Sigma\setminus\{a\}} \neg a')).$$

Let R be a regular expression and let $\mathcal{L}(R)$ be the language it defines. R can be given a tree structure, whose leaves and internal nodes belong to Σ and $\{+, \cdot, ^*\}$, respectively. Each (sub)tree identifies a (sub)expression of R. Let $e_1, .., e_n$ be the n distinct (sub)expressions of R, including elements in Σ. For each e_i, we introduce a distinct proposition letter $expr_i$. We assume $\{expr_1, .., expr_n\} \cap \Sigma = \emptyset$. Unlike proposition letters in Σ, $expr_1, .., expr_n$ are not constrained to hold over unit-intervals only. In addition, for each e_i, we introduce an auxiliary proposition letter $expr_i^{end}$ such that (i) $expr_i^{end}$ holds only at point-intervals, and (ii) if $expr_i$ holds over an interval, then $expr_i^{end}$ holds at the right endpoint of such an interval and it does not hold at any point-interval strictly included in it. For $i = 1, .., n$, the relationship between $expr_i$ and $expr_i^{end}$ is forced by the following formula:

$$\psi^{end}_{expr_i} = [G]((expr_i^{end} \rightarrow \pi) \wedge (expr_i \rightarrow (\langle A\rangle expr_i^{end} \wedge \neg\langle B\rangle\langle A\rangle expr_i^{end}))).$$

Finally, for $i = 1, .., n,$, we constrain the relationships between $expr_i$-intervals (intervals over which $expr_i$ holds) by the following formula:

$$\psi^{\not\subset}_{expr_i} = [G](expr_i \rightarrow [B]\neg expr_i \wedge \neg\langle B\rangle\langle A\rangle expr_i).$$

For any $expr_i$-interval $[i, j]$, $\psi^{\not\subset}_{expr_i}$ prevents any other $expr_i$-interval to start at k, with $i \leq k < j$. $\psi^{\not\subset}_{expr_i}$ will play a crucial role in the encoding of $\cdot$ and *.

Let us assume $e_1, .., e_n$ to be ordered according to their complexity (with $e_n = R$). The formula ψ_R is built as follows. First, we state that $expr_n$ holds over an initial interval $[0, j]$. Then, we encode the relationships between the various (sub)expressions in terms of relationships between the corresponding proposition letters. Such an encoding is inductively defined as follows.

Base case. If $e_i = a$, for some $a \in \Sigma$, we put $\psi_{expr_i} = [G]\,(expr_i \leftrightarrow a)$.

Inductive case.

Let $e_i = e_j + e_k$. We constrain the relationships between $expr_i$, $expr_j$, and $expr_k$ by means of the $AB\bar{B}$-formula $\psi_{expr_i} = [G]\,(expr_i \rightarrow expr_j \vee expr_k)$.

Let $e_i = e_j \cdot e_k$. We distinguish among three cases, namely, (i) both strings are equal to the empty string ϵ, (ii) e_j may be equal to ϵ, while e_k is not, and (iii) e_k is equal to ϵ. The three disjuncts of the consequent of the implication in the following $AB\bar{B}$-formula capture the three above cases:

$$\begin{aligned}\psi_{expr_i} = {} & [G](expr_i \rightarrow (expr_j \wedge expr_k) \vee \langle B\rangle(expr_j \wedge \langle A\rangle(expr_k \wedge \langle A\rangle expr_i^{end})) \\ & \vee(expr_j \wedge \langle A\rangle(\pi \wedge expr_k))).\end{aligned}$$

Notice that, since a given e_i may occur multiple times in R, the corresponding proposition letter $expr_i$ may be involved in more than one formula ψ_{expr_h}. As a consequence, $expr_i$ can belong to the valuation of various intervals of the model.

Formula $\psi^{\not\sqcap}_{expr_i}$ forces these intervals not to overlap: either they meet each other or they are disjoint. Such a constraint guarantees that $expr^{end}_i$ (in the second disjunct of the consequent of the above implication) is correctly associated with the right endpoint of the interval over which $expr_i$ is interpreted.

Let $e_i = e^*_j$. We distinguish three cases depending on the number of occurrences of e_j, namely, zero, one, or more than one. The three disjuncts of the consequent of the implication in the following $AB\bar{B}$-formula capture the three above cases:

$$\psi_{expr_i} = [G](expr_i \rightarrow \pi \vee expr_j \vee (\langle B\rangle(\neg\pi \wedge expr_j) \wedge \\ [B](\langle A\rangle expr^{end}_j \rightarrow \langle A\rangle(\neg\pi \wedge expr_j)) \wedge \langle A\rangle expr^{end}_j)).$$

Let us consider the third disjunct of the consequent of the outermost implication, that deals with the case of sequences of two or more occurrences of e_j. Since the $expr_i$-interval $[i, j]$ is finite and any (not punctual) $expr_j$-interval starts at a point k, with $i \leq k < j$, finiteness of the sequence of $expr_j$-intervals immediately follows. The last conjunct, together with formula $\psi^{\not\sqcap}_{expr_j}$, constrains the last element of the sequence to end at j.

Let ψ_R be the following formula:

$$\psi_R = expr_n \wedge [A]\bot \wedge \bigwedge_{1\leq i\leq n} \psi_{expr_i} \wedge \bigwedge_{1\leq i\leq n} \psi^{end}_{expr_i} \wedge \bigwedge_{1\leq i\leq n} \psi^{\not\sqcap}_{expr_i}.$$

Theorem 3. *Let R be a regular expression over Σ. $\mathcal{L}(R)$ is equal to the set of finite interval models of the $AB\bar{B}$ formula $\psi_R \wedge \psi_\Sigma$ restricted to unit-intervals and proposition letters in Σ.*

Proof. (sketch) Let $\mathbf{M} = \langle\mathbb{I}(N), A, B, \bar{B}, V\rangle$, with $N < \omega$, be an interval model, and let $w^\Sigma_{\mathbf{M}}$ be the word obtained from $\mathbf{M}$ by restricting it to unit-intervals and proposition letters in Σ, that is, $\forall a \in \Sigma\ \forall i \leq N(w^\Sigma_{\mathbf{M}}[i] = a \leftrightarrow \Sigma \cap V([i, i+1]) = \{a\})$. We first show that $\mathbf{M}, [0, N] \models \psi_R \wedge \psi_\Sigma$ iff $w^\Sigma_{\mathbf{M}} \in \mathcal{L}(R)$. Then, we show that, for each $w \in \mathcal{L}(R)$, there exists an interval model $M = \langle\mathbb{I}(N), A, B, \bar{B}, V\rangle$ such that $\mathbf{M}, [0, N] \models \psi_R \wedge \psi_\Sigma$ and $w^\Sigma_{\mathbf{M}} = w$. $\square$

It immediately follows that $\psi_R \wedge \psi_\Sigma$ is satisfiable iff $\mathcal{L}(R) \neq \emptyset$.

The encoding for regular expressions can be lifted to ω-regular ones. Let O be an ω-regular expression and let $\mathcal{L}(O)$ be the language it defines. As it happens with regular expressions, O can be given a finite tree structure. Each path from the root to a leaf in such a tree can be split into a prefix, whose nodes are ω-constructors in $\{+, \cdot,^\omega\}$, and a suffix, where internal nodes are constructors of regular expressions from the set $\{+, \cdot,^*\}$ and the leaf belongs to Σ. The encoding of ω-regular expressions O into $AB\bar{B}$ formulas is inductively defined as follows. The base case is that of regular subexpressions $e_1, .., e_n$ of O, that we already worked out. We only need to specify how to handle ω-constructors.

Let $\omega_i = \omega_j + \omega_k$, with ω_j, ω_k ω-regular expressions. The $AB\bar{B}$ formula ψ_{ω_i} for ω_i is obtained from those for ω_j and ω_k as follows: $\psi_{\omega_i} = \psi_{\omega_j} \vee \psi_{\omega_k}$.

Let $\omega_i = e_j \cdot \omega_k$, with e_j regular expression and ω_k ω-regular one. ψ_{ω_i} is defined in terms of $expr_j$ and ψ_{ω_k} as follows: $\psi_{\omega_i} = expr_j \wedge \langle A\rangle\psi_{\omega_k}$.

Let $\omega_i = e_j^\omega$, with e_j regular expression. ψ_{ω_i} is obtained from $expr_j$ as follows:

$$\psi_{\omega_i} = expr_j \wedge \langle A\rangle(\neg\pi \wedge expr_j) \wedge [\bar{B}](\langle A\rangle expr_j^{end} \to \langle A\rangle(\neg\pi \wedge expr_j)).$$

Such a formula forces the existence of an infinite sequence of $expr_j$-intervals. Uniqueness of such a sequence immediately follows from $\psi^{\not\nearrow}_{expr_j}$.

Let ψ_O be the following formula:

$$\psi_O = exp(O) \wedge \bigwedge_{1\leq i\leq n} \psi_{expr_i} \wedge \bigwedge_{1\leq i\leq n} \psi^{end}_{expr_i} \wedge \bigwedge_{1\leq i\leq n} \psi^{\not\nearrow}_{expr_i},$$

where $exp(O)$ is the $AB\bar{B}$ formula for O obtained by iterating the application of the above-defined rules for the encoding of ω-constructors in $AB\bar{B}$ until maximal regular subexpressions of O are reached (a regular subexpression of O is maximal if it is not a subexpression of any other regular subexpression).

Theorem 4. *Let O be an ω-regular expression over Σ. $\mathcal{L}(O)$ is equal to the set of interval models of the $AB\bar{B}$ formula $\psi_O \wedge \psi_\Sigma$, interpreted over $\mathbb{N}$, restricted to unit-intervals and proposition letters in Σ.*

Proof. (sketch) As in the case of Theorem 3, we first show that, for every interval model $\mathbf{M} = \langle \mathbb{I}(\omega), A, B, \bar{B}, V\rangle$, $\mathbf{M}, [0, N] \models \psi_O \wedge \psi_\Sigma$, for some $N < \omega$, iff $w^\Sigma_{\mathbf{M}} \in \mathcal{L}(O)$; then, we show that, for each $w \in \mathcal{L}(O)$, there exists an interval model M such that $\mathbf{M}, [0, N] \models \psi_O \wedge \psi_\Sigma$, for some $N < \omega$, and $w^\Sigma_{\mathbf{M}} = w$. $\square$

It immediately follows that $\psi_O \wedge \psi_\Sigma$ is satisfiable iff $\mathcal{L}(O) \neq \emptyset$.

4 From ω-Regular to ωB-Regular Languages and beyond

In this section, we provide an encoding of ωB-regular expressions, that properly extend ω-regular ones with boundedness [4], in $AB\bar{B}\bar{A}$. Moreover, we show that some natural variants of boundedness can also be dealt with by $AB\bar{B}\bar{A}$.

ωB-regular expressions are obtained from ω-regular ones by introducing a variant of Kleene star $(.)^*$, denoted by $(.)^B$, to be used in the scope of the ω-constructor $(.)^\omega$. The bounded exponent B allows one to constrain the argument R of the regular expression R^B to be repeated a bounded number of times in any ω-iteration, where the bound is fixed for the whole ω-word. As an example [4], the expression $(a^B b)^\omega$ denotes the language of ω-words for which there is an upper bound on the number of consecutive occurrences of a. As the bound may vary from word to word, the language is not ω-regular. ωB-regular expressions O over Σ are defined as follows: $O ::= B{+}B \mid R{\cdot}B \mid R'^\omega$, where R is a regular expression and R' is defined as follows: $R' ::= a \mid R' + R' \mid R' \cdot R' \mid R'^* \mid R'^B$, with $a \in \Sigma$. It is possible to show that each ωB-regular language $\mathcal{L}$ can be expressed by a suitable ωB-regular expression, and vice versa.

Unlike the case of (ω-)regular expressions, we associate a distinct proposition letter with each (sub)expression of O, apart from elements in Σ (if the same non-atomic subexpression occurs multiple times, we associate a distinct letter to each occurrence). Apart from that, the encoding of ωB-regular expressions

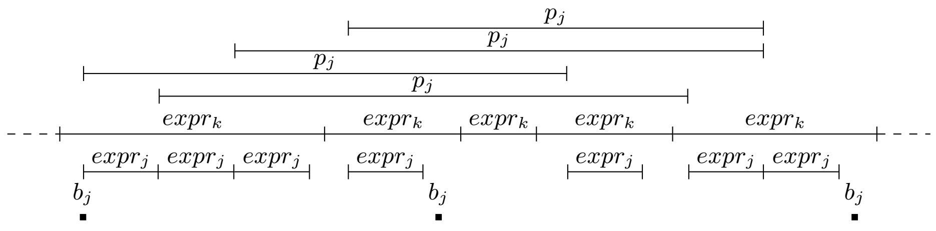


Fig. 2. The interplay of b_j- and p_j-intervals

is the same as that of ω-regular ones, but for the case of subexpressions of the form $e_i = e_j^B$. Let O be an ωB-regular expression. Wlog, we assume that for each subexpression $e_i (= e_j^B)$ of O there is a subexpression e_k^ω of O such that e_i is a proper subexpression of e_k (it is trivial to check that $(e_j^B)^\omega$ is equivalent to e_j^ω). The encoding of $e_i = e_j^B$ consists of two formulas. The first one (the local one) is the same as that for $e_i = e_j^*$; the second one (the global one) is:

$$\begin{aligned}\psi^B_{expr_j} = {} & \langle S\rangle(expr_j \wedge [A]\neg expr_j \wedge [A][A]\neg expr_j) \vee ([G](b_j \rightarrow \pi) \wedge [G](expr_j \rightarrow \\ & (\langle \bar{B}\rangle p_j \wedge [\bar{B}](p_j \rightarrow [\bar{B}]\neg p_j)) \wedge expr_j \rightarrow [B](\neg\pi \rightarrow [A]\neg b_j) \wedge \\ & p_j \rightarrow (\langle B\rangle\langle A\rangle b_j \wedge [B](\langle A\rangle b_j \rightarrow [B][A]\neg b_j)) \wedge p_j \rightarrow \langle A\rangle expr_j \wedge \\ & (expr_j \wedge \langle \bar{A}\rangle\langle \bar{A}\rangle b_j) \rightarrow \langle \bar{A}\rangle p_j \wedge p_j \rightarrow \langle B\rangle\langle A\rangle expr_k)),\end{aligned}$$

where $\langle S\rangle\varphi$ is a shorthand for $\neg[G]\neg\varphi$. The first disjunct deals with ω-words with a finite number of instances of the bounded subexpression e_j, the second with ω-words with an infinite number of instances of e_j. In the latter case, it is possible to select an infinite number of point-intervals, called *break points*, in such a way that, for every $i \geq 1$, the number of instances of e_j in between the i-th and the $(i+1)$-th break point (i-th b_j-window) is greater than or equal to the number of instances of e_j in the $(i+1)$-th b_j-window. Proposition letter b_j is used to label break points. The first conjunct forces b_j to hold at point-intervals only. The second one suitably connects $expr_j$-intervals belonging to consecutive b_j-windows by using proposition letter p_j. It states that (i) each $expr_j$-interval initiates one and only one p_j-interval; (ii) each $expr_j$-interval does not (strictly) include any b_j-interval; (iii) each p_j-interval includes one and only one b_j-interval, i.e., its endpoints belong to two consecutive b_j-windows; (iv) each p_j-interval meets an $expr_j$-interval, thus establishing a connection between the left endpoints of two $expr_j$-intervals belonging to two consecutive b_j-windows; (v) each $expr_j$-interval, which is preceded by a b_j-interval, i.e., it belongs to a b_j-window, is met by a p-interval; (vi) each p_j-interval has a nonempty intersection with (at least) two $expr_k$-intervals (ω-iterations), i.e., the endpoints of any p_j-interval belong to two different $expr_k$-intervals (ω-iterations).

The interplay of b_j- and p_j-intervals can be illustrated as follows (a graphical account is given in Figure 2). For any pair of consecutive b_j-windows W, W', p_j-intervals define a suitable mapping of the left endpoints of $expr_j$-intervals included in W to those of $expr_j$-intervals included in W'. More precisely, let $[i, j], [j, k]$ be two consecutive b_j-windows. Formula $\psi^B_{expr_j}$ forces the existence of a total, surjective mapping f from the set $\{l : \exists l'(i \leq l < l' \leq j \wedge [l, l']$ is an $expr_j$-interval$)\}$ onto the set $\{l : \exists l'(j \leq l < l' \leq k \wedge [l, l']$ is an $expr_j$-interval$)\}$. Moreover, for every p_j-interval $[l, f(l)]$, there is an $expr_k$-interval $[m, m']$ with $l \leq m < f(l)$. From surjectiveness of f, it follows that the number of $expr_j$-intervals does not increase moving from one b_j-windows to the next one. Boundedness is achieved by pairing such a condition with that preventing f from connecting points belonging to the same $expr_k$-interval (ω-iteration).

Theorem 5. *Let O be an ωB-regular expression over Σ and let $Sub_B(O)$ be the set of all subexpressions e_j of O such that e_j^B occurs in O. $\mathcal{L}(O)$ is equal to the set of interval models of the $AB\bar{B}$ formula $\psi_O \wedge \psi_\Sigma \wedge \bigwedge_{e_j \in Sub_B(O)} \psi^B_{expr_j}$, interpreted over $\mathbb{N}$, restricted to unit-intervals and proposition letters in Σ.*

Proof. ($\Rightarrow$) Let O' be the ω-regular expression obtained from O by replacing each e^B by e^*. For each ω-word w, $w \in \mathcal{L}(O)$ implies $w \in \mathcal{L}(O')$. By Th. 4, there is a model $\mathbf{M}$ such that $\mathbf{M}, [0, N] \models \psi_{O'} \wedge \psi_\Sigma$, for some $N < \omega$, and $w^\Sigma_{\mathbf{M}} = w$. Since $w \in \mathcal{L}(O)$, for each $e_j \in Sub_B(O)$, where e_j^B is a proper subexpression of some e_k such that e_k^ω is a (sub)expression of O, there is $max_j \in \mathbb{N}$ such that the maximum number of $expr_j$-intervals included in an $expr_k$-interval is equal to max_j. A model $\mathbf{M}'$ for $(\psi_O \wedge \psi_\Sigma \wedge) \bigwedge_{e_j \in Sub_B(O)} \psi^B_{expr_j}$ can be obtained from $\mathbf{M}$ by extending its valuation function V to b_j and p_j, for each $e_j \in Sub_B(O)$ (we only consider the case of ω-words with an infinite number of instances of e_j). We select a point k such that there are exactly max_j points l before k such that $expr_j \in V([l, m])$, for some $m > l$. Then, we take an infinite sequence of points $k_0 = 0 < k_1 = k < k_2 < ..$ such that, for every $i \geq 0$, $|\{l : k_i < l < k_{i+1} \wedge \exists l'(l' > l \wedge expr_j \in V([l, l']))\}| = max_j$ and we insert b_j into $V'([k_i, k_i])$, for every $i \geq 0$. Now, for every $i \geq 0$, let $k_i \leq l_1 < l_2 < .. < l_{max_j} < k_{i+1} \leq \hat{l}_1 < \hat{l}_2 < .. < \hat{l}_{max_j} < k_{i+2}$ be such that, for $1 \leq m \leq max_j$, $expr_j \in V[l_m, l']$, for some $l' > l_m$, and $expr_j \in V[\hat{l}_m, l'']$, for some $l'' > \hat{l}_m$. We insert p_j into $V([l_m, \hat{l}_m])$, for each $1 \leq m \leq max_j$. To show that, for each $1 \leq m \leq max_j$, the endpoints of the p_j-interval $[l_m, \hat{l}_m]$ belong to two different $expr_k$-intervals, it suffices to observe that there are exactly max_j distinct points in between l_m and $\hat{l}_m$ that start an $expr_j$-interval. Thus, assuming that both l_m and $\hat{l}_m$ belong to the same $expr_k$-interval would lead to the conclusion that such an $expr_k$-interval contains $max_j + 1$ $expr_j$-intervals (contradiction).

($\Leftarrow$) Let $\mathbf{M}$ be a model for $\psi_O \wedge \psi_\Sigma \wedge \bigwedge_{e_j \in Sub_B(O)} \psi^B_{expr_j}$. Since $\mathbf{M}$ satisfies $\psi_O \wedge \psi_\Sigma$, by Th. 4, it follows that $w^\Sigma_{\mathbf{M}} \in \mathcal{L}(O')$, where O' is obtained from O by replacing each e^B by e^*. We must show that $w^\Sigma_{\mathbf{M}}$ conforms to e_j^B as well, for each $e_j \in Sub_B(O)$. We restrict our attention to the case in which there is an infinite number of $expr_j$-intervals, and thus of b_j-intervals, in $\mathbf{M}$ (the other case is trivial). Since $\mathbf{M}$ satisfies $\bigwedge_{e_j \in Sub_B(O)} \psi^B_{expr_j}$, for each $e_j \in Sub_B(O)$, there is a least $max_j \in \mathbb{N}$ such that the number of $expr_j$-intervals included in a b_j-window is less than or equal to it. Boundedness easily follows. Suppose, by contradiction, that this is not the case for some $e_j \in Sub_B(O)$ subexpression of e_k^ω, i.e., for every $N \in \mathbb{N}$ there are l, m such that $expr_k \in V([l,m])$ and the number of $expr_j$-intervals included in $[l,m]$ is greater than N. Now, let $N > 3 \cdot max_j$ and let $[l,m]$ be an $expr_k$-interval that includes more than N $expr_j$-intervals. By definition of max_j, there are n, n', and n'', with $l \leq n < n' < n'' \leq m$, such that $b_j \in V([n,n]) \cap V([n',n']) \cap V([n'',n''])$. It follows that there are r, s, with $n \leq r < n' \leq s < n''$, such that $p_j \in V([r,s])$. Now, $expr_k \in V([l,m])$ and $\psi^{\not\emptyset}_{expr_k}$ prevents any other $expr_k$-interval to start at some t, with $l \leq t < m$. Since $l \leq r < s < m$, this implies that the conjunct $p_j \rightarrow \langle B\rangle\langle A\rangle expr_k$ is not satisfied and thus $\mathbf{M}$ is not a model for $\psi_O \wedge \psi_\Sigma \wedge \bigwedge_{e_j \in Sub_B(O)} \psi^B_{expr_j}$ (contradiction). □

It directly follows that $\psi_O \wedge \psi_\Sigma \wedge \bigwedge_{e_j \in Sub_B(O)} \psi^B_{expr_j}$ is satisfiable iff $\mathcal{L}(O) \neq \emptyset$.

We introduce now two variants of boundedness, namely, strict and weak monotonicity. The latter one can be imposed to both regular and ω-regular languages, the former to regular ones only. Strictly-monotonic regular ($<$-*regular*) languages are obtained from regular ones by introducing a variant of Kleene star $(.)^*$, denoted by $(.)^<$, to be used in the scope of $(.)^*$. By analogy with ωB-regular languages, wlog we can assume that, for each subexpression of the form $e_i = e_j^<$ of a given expression R, there is a (sub)expression e_k^* of R such that e_i is a proper subexpression of e_k ($(e_j^<)^*$ is equivalent to e_j^*). For any pair of consecutive occurrences of e_k, $(.)^<$ constrains the number of occurrences of e_j included in the first one to be *greater than* the number of those included in the second one. Unlike that of $(.)^B$, the encoding of $(.)^<$ does not require the use of proposition letters b_j, as we can restrict our attention to pairs of consecutive occurrences of e_k only. Weakly-monotonic regular ($\leq$-*regular*) languages are obtained by substituting $(.)^\leq$ for $(.)^<$. The only difference between $(.)^\leq$ and $(.)^<$ is the replacement of condition '*greater than*' by '*greater than or equal to*'. The encoding of $e_i = e_j^<$ (resp., $e_i = e_j^\leq$) consists of a local and a global formula. The former one is the same as that for $e_i = e_j^*$; the latter one is the formula $\psi^<_{expr_j}$ (resp., $\psi^\leq_{expr_j}$):

$$\begin{aligned}
\psi^<_{expr_j} = {} & [G]((expr_j \wedge \langle A\rangle expr_j \wedge \langle A\rangle\langle A\rangle expr_k) \rightarrow \\
& (((\langle \bar{B}\rangle p_j \wedge [\bar{B}](p_j \rightarrow [\bar{B}]\neg p_j)) \vee [A][A](expr_k \rightarrow [B][A]\neg expr_j)) \\
& \wedge\ (expr_j \wedge [A]\neg expr_j) \rightarrow [\bar{B}]\neg p_j \wedge (expr_j \wedge \langle \bar{A}\rangle\langle \bar{A}\rangle expr_k) \rightarrow \langle \bar{A}\rangle p_j \\
& \wedge\ p_j \rightarrow (\langle B\rangle\langle A\rangle expr_k \wedge [B](\langle A\rangle expr_k \rightarrow [B][A]\neg expr_k))); \\
\psi^\leq_{expr_j} = {} & [G]((expr_j \wedge \langle A\rangle\langle A\rangle expr_k) \rightarrow (((\langle \bar{B}\rangle p_j \wedge [\bar{B}](p_j \rightarrow [\bar{B}]\neg p_j)) \vee \\
& [A][A](expr_k \rightarrow [B][A]\neg expr_j)) \wedge\ (expr_j \wedge \langle \bar{A}\rangle\langle \bar{A}\rangle expr_k) \rightarrow \langle \bar{A}\rangle p_j \\
& \wedge\ p_j \rightarrow (\langle B\rangle\langle A\rangle expr_k \wedge [B](\langle A\rangle expr_k \rightarrow [B][A]\neg expr_k))).
\end{aligned}$$

Theorem 6. *Let R be a $<$-regular (resp., $\leq$-regular) expression over Σ and let $Sub_<(R)$ (resp., $Sub_\leq(R)$) be the set of all subexpressions e_j of R such that $e_j^<$ (resp., $e_j^\leq$) occurs in R. $\mathcal{L}(R)$ is equal to the set of finite interval models of the $AB\bar{B}$ formula $\psi_R \wedge \psi_\Sigma \wedge \bigwedge_{e_j \in Sub_<(R)} \psi^<_{expr_j}$ (resp., $\bigwedge_{e_j \in Sub_\leq(R)} \psi^\leq_{expr_j}$) restricted to unit-intervals and proposition letters in Σ.*

Weakly-monotonic ω-regular ($\omega\leq$-*regular*) languages are obtained from ω-regular ones by adding the operator $(.)^{<}$ to be used in the scope of $(.)^{\omega}$. The addition of $(.)^{\leq}$ can be dealt with as in regular languages, the only difference being that the second conjunct of subformula $expr_j \wedge \langle A\rangle\langle A\rangle expr_k$ of $\psi^{\leq}_{expr_j}$ can be removed.

Theorem 7. *Let O be an $\omega\leq$-regular expression over Σ and let $Sub_{\leq}(O)$ be the set of all subexpressions e_j of O such that $e_j^{\leq}$ occurs in O. $\mathcal{L}(O)$ is equal to the set of interval models of the $AB\bar{B}$ formula $\psi_O \wedge \psi_{\Sigma} \wedge \bigwedge_{e_j\in Sub_{\leq}(O)} \psi^{\leq}_{expr_j}$, interpreted over $\mathbb{N}$, restricted to unit-intervals and proposition letters in Σ.*

5 A Decidable Fragment of $AB\bar{B}\bar{A}$ over $\mathbb{N}$

In this section, we show that (i) decidability over $\mathbb{N}$ can be recovered by restricting the set of models (compass generators) over which $AB\bar{B}\bar{A}$ formulas are interpreted, and (ii) the resulting class of models is expressive enough to encode the (non)emptiness problem for extended ω-regular languages.

Let φ be an $AB\bar{B}\bar{A}$ formula and $\mathcal{G} = (\mathbb{P}(N), A, \bar{A}, B, \bar{B}, \mathcal{L})$, with $N \leq \omega$, be a compass structure. The notion of *cover* $\mathcal{C}$ and the counter functions $\mathcal{S}hading$ and $\mathcal{C}ount$ are defined as follows [12]. A *cover* $\mathcal{C}$ for $\mathcal{G}$ is a minimal subset of N such that, for each $0 \leq y \leq N$ and $\psi \in \mathcal{R}eq_{\overline{A}}(\mathcal{L}(y,y))$, there is $x \in \mathcal{C}$ with $\psi \in \mathcal{L}(x,y)$. For each $F \in \mathcal{A}_{\varphi}$ and $0 \leq y \leq N$, the functions $\mathcal{S}hading : \mathcal{A}_{\varphi} \times N \to \mathbb{N}$ and $\mathcal{C}ount : \mathcal{A}_{\varphi} \times N \to \mathbb{N}$ return the values $\mathcal{S}hading(F,y) = |\{x : \mathcal{L}(x,y) = F\}|$ and $\mathcal{C}ount(F,y) = |\{x : \mathcal{L}(x,y) = F \wedge x \in \mathcal{C}\}|$, respectively.

Definition 8. *Let φ be an $AB\bar{B}\bar{A}$ formula, $\mathcal{G} = (\mathbb{P}(N), A, \bar{A}, B, \bar{B}, \mathcal{L})$, with $N < \omega$, be a consistent (but not necessarily fulfilling) finite compass structure that features φ, $\mathcal{C}$ be a cover for $\mathcal{G}$, and $\overline{y} < N$. We say that the triple $\overline{\mathcal{G}} = (\mathcal{G}, \overline{y}, \mathcal{C})$ is a compass generator iff (i) for each $F \in \mathcal{A}_{\varphi}$, $\mathcal{S}hading(F,N) > 0$ iff $\mathcal{S}hading(F,\overline{y}) > 0$ and $\mathcal{C}ount(F,\overline{y}) \leq \mathcal{S}hading(F,N)$, and (ii) for each $0 \leq x \leq \overline{y}$ and each $\psi \in \mathcal{R}eq_{\overline{B}}(\mathcal{L}(x,\overline{y}))$, there is $\overline{y} < y' \leq N$ such that $\psi \in \mathcal{L}(x,y')$.*

Theorem 9. *An $AB\bar{B}\bar{A}$ formula φ is satisfiable in $\mathbb{N}$ iff there is a compass generator $\overline{\mathcal{G}} = (\mathcal{G}, \overline{y}, \mathcal{C})$ for it.*

One can (easily) build a consistent and fulfilling ultimately periodic compass structure $\mathcal{G}' = (\mathbb{P}(\omega), A, \bar{A}, B, \bar{B}, \mathcal{L}')$ for φ starting from a compass generator $\overline{\mathcal{G}} = (\mathcal{G}, \overline{y}, \mathcal{C})$ for it [12], thus showing that if φ has a model over $\mathbb{N}$, then it has an ultimately periodic one. Unfortunately, the problem of establishing the existence of a compass generator for an $AB\bar{B}\bar{A}$ formula φ is in general undecidable. We now introduce a restricted class of compass generators.

Definition 10. *Let φ be an $AB\bar{B}\bar{A}$ formula and let $\overline{\mathcal{G}} = (\mathcal{G}, \overline{y}, \mathcal{C})$, with $\mathcal{G} = (\mathbb{P}(N), A, \bar{A}, B, \bar{B}, \mathcal{L})$, be a compass generator for it. We say that $\overline{\mathcal{G}}$ is* contraction-free *if for every y, y', with $0 < y < y' < \overline{y}$ or $\overline{y} < y < y' < N$, there is $F \in \mathcal{A}_{\varphi}$ such that $\mathcal{C}ount(F,y) \not\leq \mathcal{C}ount(F,y')$.*

We say that an $AB\bar{B}\bar{A}$ formula is *contraction-free satisfiable* if there exists a contraction-free compass generator $\overline{\mathcal{G}} = (\mathcal{G}, \overline{y}, \mathcal{C})$ for it.

Theorem 11. *Let φ be an $AB\bar{B}\bar{A}$ formula. The problem of establishing the existence of a contraction-free compass generator $\overline{\mathcal{G}} = (\mathcal{G}, \overline{y}, \mathcal{C})$ for it is decidable.*

Theorem 12. *Let O be an ωB-regular (resp., $\omega\leq$-regular) expression over Σ and let $Sub_B(O)$ (resp., $Sub_{\leq}(O)$) be the set of all subexpressions e_j of O such that e_j^B (resp., $e_i^{\leq}$) occurs in O. It holds that $\psi_O \wedge \psi_\Sigma \wedge \bigwedge_{e_j \in Sub_B(O)} \psi^B_{expr_j}$ (resp., $\bigwedge_{e_j \in Sub_{\leq}(O)} \psi^{\leq}_{expr_j}$) is satisfiable iff it is contraction-free satisfiable.*

6 Conclusions

In this paper, we built a bridge between interval temporal logics and (extended) regular languages of finite and infinite words. Thanks to it, classical problems for extended (ω-)regular languages, such as the (non)emptiness one, can be naturally reformulated in logical terms, and new meaningful classes of languages can be defined and studied. In a companion paper, we showed that ωS-regular languages, as well as some natural variants of them, can be defined in $AB\bar{B}$ extended with an equivalence relation. We are working at a logical characterization of ωBS-regular languages in $AB\bar{B}\bar{A}$ extended with an equivalence relation.

References

1. Allen, J.: Maintaining knowledge about temporal intervals. Communications of the ACM 26(11), 832–843 (1983)
2. Bojańczyk, M.: A Bounding Quantifier. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 41–55. Springer, Heidelberg (2004)
3. Bojańczyk, M.: Weak MSO with the unbounding quantifier. Theory of Computing Systems 48(3), 554–576 (2011)
4. Bojańczyk, M., Colcombet, T.: ω-regular expressions with bounds. In: LICS, pp. 285–296. IEEE Computer Society (2006)
5. Bresolin, D., Goranko, V., Montanari, A., Sala, P.: Tableaux for logics of subinterval structures over dense orderings. J. of Logic and Comp. 20(1), 133–166 (2010)
6. Bresolin, D., Goranko, V., Montanari, A., Sciavicco, G.: Propositional interval neighborhood logics: Expressiveness, decidability, and undecidable extensions. Annals of Pure and Applied Logic 161(3), 289–304 (2009)
7. Bresolin, D., Montanari, A., Sala, P., Sciavicco, G.: What's decidable about Halpern and Shoham's interval logic? The maximal fragment $\mathsf{AB\overline{B}\overline{L}}$. In: LICS, pp. 387–396. IEEE Computer Society (2011)
8. Goranko, V., Montanari, A., Sciavicco, G.: A road map of interval temporal logics and duration calculi. J. of Applied Non-Classical Logics 14(1-2), 9–54 (2004)
9. Halpern, J., Shoham, Y.: A propositional modal logic of time intervals. J. of the ACM 38(4), 935–962 (1991)
10. Lodaya, K.: Sharpening the Undecidability of Interval Temporal Logic. In: Kleinberg, R.D., Sato, M. (eds.) ASIAN 2000. LNCS, vol. 1961, pp. 290–298. Springer, Heidelberg (2000)
11. Montanari, A., Puppis, G., Sala, P.: A Decidable Spatial Logic with Cone-Shaped Cardinal Directions. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 394–408. Springer, Heidelberg (2009)

12. Montanari, A., Puppis, G., Sala, P.: Maximal Decidable Fragments of Halpern and Shoham's Modal Logic of Intervals. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 345–356. Springer, Heidelberg (2010)
13. Montanari, A., Puppis, G., Sala, P., Sciavicco, G.: Decidability of the interval temporal logic $AB\bar{B}$ on natural numbers. In: STACS, pp. 597–608 (2010)
14. Moszkowski, B.: Reasoning about digital circuits. Tech. rep. stan-cs-83-970, Dept. of Computer Science, Stanford University, Stanford, CA (1983)
15. Venema, Y.: A modal logic for chopping intervals. J. of Logic and Comp. 1(4), 453–476 (1991)

Eliminating Stack Symbols in Push-Down Automata and Linear Indexed Grammars

Katsuhiko Nakamura and Keita Imada

School of Science and Engineering, Tokyo Denki University
Hatoyama-machi, Saitama-ken, 350-0394 Japan
nakamura@rd.dendai.ac.jp

Abstract. This paper investigates two subjects in push-down automata (PDAs) and linear indexed grammars (LIGs), which are extended PDAs, focusing on eliminating the stack symbols. One of the subjects is concerned with *PI- (push-input-)* PDA and PI-LIG without ϵ-transition rule, in which only input symbols are pushed down to the stack. It is shown that the class of languages of PI-LIGs is incomparable with that of PDAs, which is the class of context-free languages (CFLs). The other subject is a simple bottom-up parsing method for LIGs, in which the stack symbols are eliminated at the first step of the parsing. The paper shows several PI-LIGs, including PI-PDAs for fundamental context-free and context-sensitive languages, which are synthesized by a grammatical inference system LIG Learner.

Keywords: PDA, LIG, mildly context sensitive language, stack symbol, grammatical inference, bottom-up parsing.

1 Introduction

Linear indexed grammar (LIG) was introduced by Duske [3] as a restricted indexed grammar by Aho [1]. LIG in this paper is a subclass of tree-adjoining grammars (TAGs) [6,8] and closely related to some other types of grammars in a class known as mildly context-sensitive grammars [7]. These grammars have the following common features: the class of the languages includes not only all context-free languages, but also several fundamental context-sensitive languages; and the parsing is within polynomial time.

Although LIG is called a grammar, it is essentially an extended push-down automaton (PDA), which is also called right-linear right-indexed (RIR) grammar in [1]. Generally, automata for accepting languages are distinguished from grammars for deriving languages. However, we can define a language by a set of rules, which can be used for not only accepting but also deriving strings of the language. Therefore, we identify PDA with grammars.

In this paper, we investigate two subjects on PDAs and LIGs, focusing on eliminating the stack symbols. One of the subjects is concerned with *PI- (push-input-)* PDAs and LIGs without ϵ-transition rule, in which only input symbols are pushed down to the stack. The other subject is the bottom-up parsing, in which the stack symbols are eliminated at the first step of the parsing. This parsing method is simpler than the other methods previously published such as in [2,8].

The motivation for this work is derived from the grammatical inference of LIGs. The rules in push-input form have no special stack symbols, reducing the computation time to generate the rules and to search for any sets of rules for a language. The simple bottom-up parsing is essential for incremental learning using the bridging rule generation implemented in Synapse system [9], which is based on the results of bottom-up parsing of positive samples.

There have been several works on extension and restriction of PDAs and their relations to CFLs and mildly context-sensitive languages. Goldstine [5] showed that two stack symbols in PDAs are sufficient to accept any context-free language (CFL). Although the push-input type PDAs are used in many examples in most textbooks on formal languages and automata, the generality or limitation of this form have not been previously studied to the best of the authors knowledge.

This paper is organized as follows. Section 2 defines PDAs, LIGs, push-input form and the derivation of languages of these grammars. Section 3 introduces the notions of normalized LIGs and coded palindromes, and discusses the relationships between the language classes of PDAs, LIGs, PI-PDAs and PI-LIGs. Section 4 presents the simple bottom-up parsing method for LIGs, which is specified by forward inference rules. Section 5 shows several PI-PDAs and PI-LIGs for fundamental formal languages, synthesized by the grammatical inference system LIG Learner. Finally, Section 6 concludes the paper and describes future research subjects.

2 Push-Down Automata and Linear-Indexed Grammars

We represent the contents of a stack with a symbol x at the top by $[x\ z]$, where z is a substring in the rest of the stack, which may be the empty string ϵ.

A LIG is a system $G = (K, \Sigma, X, P, s, F)$, where:

- K, Σ and X are finite sets of *states*, *input symbols* and *stack symbols*, respectively, with $K \cap \Sigma = \emptyset$;
- P is a finite set of *rules* of the forms either *forward-input rule*: $p[x] \to a\, q[\tau]$, *backward-input rule*, $p[x] \to q[\tau]\ a$ or *ϵ-transition rule*: $p[x] \to q[\tau]$, where for $a, b \in \Sigma,\ \tau \in X \cup \{\epsilon\}$ and $x \in X \cup \{\epsilon\}$;
- $s \in K$ is an *initial state*; and
- $F \subseteq K$ is a set of *final states.*

A *push-down automaton* (*PDA*) is a LIG without backward-input rules. We represent any term $p[\,] = p[\epsilon]$ in the rules simply by the state p, e.g., $p[f] \to a\, q$ is equivalent to $p[f] \to a\, q[\,]$, and $p \to q\, a$ is $p[\,] \to q[\,]a$. We call a state p the *pushing state*, if the LIG has a rule of the form $p \to aq[f]$, the *pop-up state*, if it has $p[f] \to aq$, and the *simple state* otherwise.

For any LIG $G = (K, \Sigma, X, P, s, F)$, the derivation relation "$\Rightarrow_G$" over $\{u\, p\, [z]\, v \mid u, v \in \Sigma^*,\ p \in K \text{ and } z \in X^*\}$ is defined by,

- $u\,p\,[x\,z]\,v \Rightarrow_G u\,a\,q\,[\tau\,z]\,v$, if $(p\,[x] \rightarrow a\,q\,[\tau]) \in P$,
- $u\,p\,[x\,z]\,v \Rightarrow_G u\,q\,[\tau\,z]\,a\,v$, if $(p\,[x] \rightarrow q\,[\tau]\;a) \in P$,
- $u\,p\,[\,]\,v \Rightarrow_G u\;v$, if $p \in F$,

for all $a, b, \in \Sigma$, $\tau \in X \cup \{\epsilon\}$ and $u, v \in \Sigma^*$. Note that the third derivation is represented by the termination rule in other formalism. The language of the grammar G is the set $L\,(G) = \{w \in \Sigma^* |\; s\;[\,]\; \Rightarrow_G^+ w\}$, where $\Rightarrow_G^+$ is the transitive closure of $\Rightarrow_G$.

We represent a class of languages of a grammar class $\mathcal{G}$ by $\mathcal{C}(\mathcal{G})$. Aho [1] and Duske [3] proved the following basic proposition.

Proposition 1. *All the languages in $\mathcal{C}(LIG)$ are derived by LIGs with the rules of the following forms (1) – (6), where p and q are states, a and b are input symbols, and c is a stack symbol. All the languages in $\mathcal{C}(PDA)$ are derived by LIGs (PDAs) with the forward-input rules of the forms (1), (2) and (3).*

(1) $p \rightarrow a\,q$ *(2)* $p \rightarrow a\,q\,[c]$ *(3)* $p\,[c] \rightarrow a\,q$
(4) $p \rightarrow q\,a$ *(5)* $p \rightarrow q[c]\,a$ *(6)* $p\,[c] \rightarrow q\,a$

The *push-input rule* of the form $p \rightarrow aq\,[a]$ pushes down the input symbol a. Any PDA and LIG are *push-input*, if all the pushing rules are push-input, and hence all the stack symbols are the input symbols. *PI-PDAs* and *PI-LIGs* are push-input PDAs and LIGs, respectively, that have no ϵ-transition rule.

2.1 Examples of LIGs

The following two PI-LIGs are for well-known non-context-free languages. They were synthesized by LIG Learner described in Section 5.

Example 1: Copy Language: The copy language, i.e., the set of strings of a's and b's with the form ww, is derived by the following six rules with the starting symbol s and the final state p.

$$s \rightarrow a\,s[a],\;\; s \rightarrow b\,s\,[b],\;\; s\,[a] \rightarrow p\,a,\;\; s\,[b] \rightarrow p\,b,\;\; p\,[a] \rightarrow p\,a,\;\; p\,[b] \rightarrow p\,b.$$

String $aabaab$ is derived by the derivation sequence,

$$s\,[\,] \;\Rightarrow a\,s\,[a] \Rightarrow\; aa\,s\,[a\,a] \Rightarrow aab\,s\,[b\,a\,a] \;\Rightarrow aab\,p\,[a\,a]\,b \;\Rightarrow aabp\,[a]\,ab$$
$$\Rightarrow aab\,p\,[\,]\,aab \;\Rightarrow aabaab.$$

Example 2: Set $\{a^n b^n c^n \mid n \geq 1\}$: This language is derived by the following four rules with the starting symbol s and the final state q.

$$s \rightarrow a\,p\,[a],\;\; s\,[a] \rightarrow b\,q,\;\; p \rightarrow s\,c,\;\; q\,[a] \rightarrow b\,q.$$

An example of derivation sequence is:

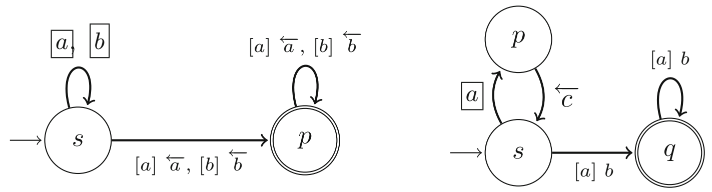
$$s\,[\,] \;\Rightarrow a\,p\,[a] \Rightarrow\; a\,s\,[a]\,c \Rightarrow aa\,p\,[a\,a]\,c \Rightarrow aa\,s\,[a\,a]\,cc \;\Rightarrow aab\,q\,[a]\,cc$$
$$\Rightarrow aabb\,q\,[\,]\,cc \;\Rightarrow aabbcc.$$

2.2 State Transition Diagrams

We represent LIGs by state transition diagrams to easily understand the structures. The diagrams are based on those for finite state automata, but extended so that each edge has, at most, the following three labels depending on the types of the rules:

- pop-up stack symbols $[f]$ for rules of the forms $p[f] \to a\,q$, or $p[f] \to q\,a$;
- an input symbol, a, $\overleftarrow{a}$ or ϵ, for a forward-input rule, a backward-input rule or an ϵ-rule, respectively; and
- a push-down stack symbol, $\boxed{g}$ for rules $p[\sigma] \to aq[g]$ or $p[\sigma] \to \epsilon\, q[g]$.

For example, the labels of the edge for a rule $p[f] \to a\,q$ are [f] and a, and the labels for $p \to q[g]\,a$ are $\overleftarrow{a}$ and $\boxed{g}$. For the case of a push-input rule $p \to a\,q[a]$, we can simply write the label $\boxed{a}$ instead of $a\,\boxed{a}$. Fig. 1 shows the state transition diagrams for Example 1 and 2.


$\boxed{a}$, $\boxed{b}$
$[a]\ \overleftarrow{a}$, $[b]\ \overleftarrow{b}$
s
p
$[a]\ \overleftarrow{a}$, $[b]\ \overleftarrow{b}$
p
$\boxed{a}$
$\overleftarrow{c}$
$[a]\ b$
s
q
$[a]\ b$

(a) Copy Language (b) Set $\{a^n b^n c^n | n \geq 1\}$

Fig. 1. State Transition Diagrams of LIGs

3 PI-PDAs and PI-LIGs

The following lists shows the forms of rules in PI-LIGs, where p and q are states, and a and c are input symbols.

(1) $p \to a\,q$ (2) $p \to a\,q\,[a]$ (3) $p\,[c] \to a\,q$
(4) $p \to q\,a$ (5) $p \to q[a]\,a$ (6) $p\,[c] \to q\,a$

The PI-PDA uses only forward-input rules of the forms (1), (2) and (3).

For any states p and q in a LIG $G = (K, \Sigma, X, P, s, F)$, q is a *combinational state* of p, if $p[x] \Rightarrow_G^* u\,q[x]\,v$ for any $u, v \in \Sigma^+, x \in X^*$, i.e., the transition from the state p to q derives substrings u and v and that the stack at state q has, or returned to, the same content x at p. For example, in the LIG for $\{a^n b^n c^n | n \geq 1\}$ in Example 2, s and q are combinational state of p and s, respectively.

3.1 Normalized LIGs

The merging of two strings w_1 and w_2 is the nondeterministic process to recursively apply the following transformation rules to $merge(w_1, w_2)$.

$$merge(\epsilon, w) \rightarrow w, merge(w, \epsilon) \rightarrow w.$$
$$merge(a_1a_2 \cdots a_m, w) \rightarrow a_1 \cdot merge(a_2 \cdots a_m, w).$$
$$merge(w, b_1b_2 \cdots b_n) \rightarrow b_1 \cdot merge(w, b_2 \cdots b_n).$$

for $m, n \geq 1$, where "$\cdot$" is the concatenation operator. For any string $w = a_1a_2 \cdots a_n$, the *renamed string* of w is the string $\overline{w} = \overline{a}_1\overline{a}_2 \cdots \overline{a}_n$.

Let $T(w)$ be the set of merged strings of w_1 and the reversal of the renamed string $\overline{w}_2^R$ with $w = w_1w_2$. Any string in $T(w)$ can be converted to w. For a string $w = a_1a_2 \cdots a_n$, $T(w)$ contains, for example, $a_1a_2 \cdots a_i\,\overline{a}_n\,\overline{a}_{n-1} \cdots \overline{a}_{i+1}$ and $a_1\overline{a}_n\,a_2\,\overline{a}_{n-1} \cdots a_i\overline{a}_{i+1}$.

Any LIG $G = (K, \Sigma, I, P, s, F)$ is transformed into the *normalized LIG* of G that is the PDA $G' = (K, \Sigma \cup \{\overline{a}|a \in \Sigma\}, I, P', s, F)$ by replacing each rule of the forms $p \rightarrow q\,a$, $\quad p \rightarrow q\,[c]\,a$ and $p\,[c] \rightarrow q\,a$ in G by the rules of the forms $p \rightarrow \overline{a}\,q$, $p \rightarrow \overline{a}\,q\,[c]$, and $p\,[c] \rightarrow \overline{a}\,q$, respectively.

For example, the normalized LIG of Example 1 for the copy language derives the merged string $ab\,\overline{b}\,\overline{a}$ for $abab$, and $aab\,\overline{b}\,\overline{a}\,\overline{a}$ for $aabaab$, which are essentially palindromes. The normalized LIG of Example 2 for the set $\{a^nb^nc^n|n \geq 1\}$ derives strings $a\,\overline{c}\,b$, $a\,\overline{c}\,a\,\overline{c}\,bb$, $a\,\overline{c}\,a\,\overline{c}\,a\,\overline{c}\,bbb$, $\cdots$. It is obvious from the transformation that the normalized LIG satisfies the following proposition.

Proposition 2. *For any LIG (or PI-LIG) G, let G' be the normalized LIG (or PI-LIG, respectively) of G. For any string w, we have $w \in L(G)$, if and only if there is a string $w' \in T(w)$ with $w' \in L(G')$.*

The normalized LIGs are used for investigating the characteristics of LIGs by comparing the LIGs with the CFLs. For example, a normalized LIG is used to show a derivation tree in Section 5.1.

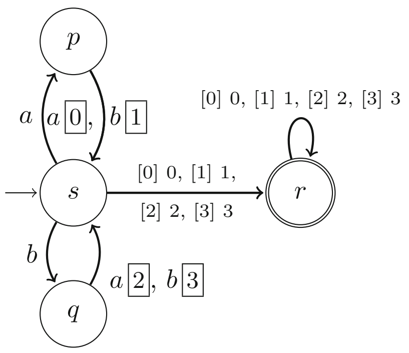
3.2 Coded Palindromes and Coded Repetitions

Let D and E be two finite sets of strings such that there is a one-to-one correspondence between two sets. A *coded palindrome* is the string $d_1d_2 \cdots d_n\,e_n \cdots e_2e_1$ and a *coded repetition* is the string $d_1d_2 \cdots d_n\;e_1e_2 \cdots e_n$ $(n \geq 1)$, where $d_1d_2 \cdots d_n$ is in D^+ and $e_1e_2 \cdots e_n$ is in E^+; and for each i, c_i corresponds to d_i.

The coded palindrome and the coded repetition are *left-coded*, if $|d_i| > |e_i| = 1$ for all $1 \leq i \leq n$. The both strings are *left-right-coded*, if either $|d_i| > |e_i| = 1$ or $|e_i| > |d_i| = 1$ for all $1 \leq i \leq n$. If D and E are the same set of symbols, the coded palindrome is a usual palindrome, and the coded repetition is the copy language. In addition, for any sets D and E the set of the coded palindromes is a CFL and that the set of the coded repetitions is a LIG language.

Example 3: Binary Coded Palindrome: The binary left-coded palindrome is the language of the CFG with one nonterminal symbol s and the following production rules,

$$s \rightarrow aas0,\; s \rightarrow abs1,\; s \rightarrow bas2,\; s \rightarrow bbs3,\; s \rightarrow \epsilon.$$



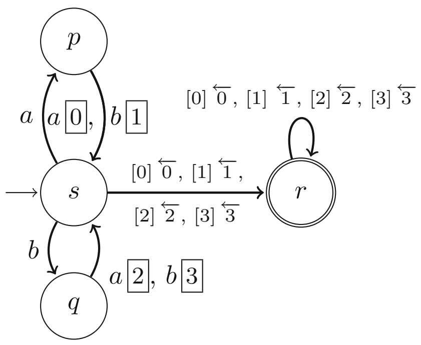


Fig. 2. State Transition Diagrams of PDA M_{BCP} for the Binary Coded Palindrome (left) and a LIG for Binary Coded Repetition (right)

This language contains strings,

$$aa0,\ ab1,\ ba2,\ bb3, aaaa00, aaab01, aaba02, aabb03, abaa10, abab11, abbba12, \cdots .$$

The language is derived by PDA M_{BCP} with the state-transition diagram in Fig. 2 (left), in which all push-down rules are not in push-input form. The following sequence shows an example derivation of a binary coded palindrome.

$$s[\,] \Rightarrow ap[\,] \Rightarrow abs\,[1] \Rightarrow abbq[1] \Rightarrow abbas\,[2\,1] \Rightarrow abba2r[1] \Rightarrow abba21r[\,].$$

This PDA is transformed into a LIG for the binary coded repetition by changing the eight left-input pop-up rules above into the backward-input pop-up rules rules as shown in Fig. 2 (right).

Proposition 3. *There is no PI-PDA G that derives the binary left-coded palindromes.*

This proposition implies that $\mathcal{C}(\text{PI-PDA}) \subsetneq \mathcal{C}(\text{PDA}) = \mathcal{C}(\text{CFG})$.

Proof. Suppose that there is a PI-PDA G that derives the binary-coded palindromes in Example 3. Then, G has a derivation sequence,

$$s[\,] \Rightarrow^* d_1 d_2 \cdots d_n\ p[x] \Rightarrow^* d_1 d_2 \cdots d_n\ e_n \cdots e_2 e_1 r[\,],$$

for a state p and a final state r, where $d_i \in \{aa, ab, ba, bb\}$ and $e_i \in \{0, 1, 2, 3\}$ for all $1 \le i \le n$. As G has no ϵ-transition rule, and the stack is empty after G derives the string $e_1 e_2 \cdots e_n$, the length of the stack is restricted to n. Hence, as all the push-down rule in G are push-input, the stack can have at most the strings in $\{a, b\}^n$. Each of the symbols in the strings needs to correspond to $d_i, 1 \le i \le n$. However, the size $|\{0, 1, 2, 3\}^n|$ increases by 4^n, whereas the size $|\{a, b\}^n|$ increases by 2^n. This causes a contradiction. □

Note that any set of the reversals of the left-coded palindromes, i.e., the right-coded palindromes, is a PI-PDA language. As $\mathcal{C}$(PI-LIG) is closed under reversal, the set of the binary left-coded palindromes is a PI-LIG language.

Proposition 4. *$\mathcal{C}$(PI-LIG) is incomparable with $\mathcal{C}$(PDA).*

In the proof of this proposition, we use the following two lemmas. Lemma 1 is based on Theorem 2.8 by Duske [3], in which the PI-LIG and PI-PDA are simply LIG and PDA (equivalently, CFG), respectively. The proof is similar to that of this original theorem. Lemma 2 is a corollary of Proposition 3.

Lemma 1. *For any alphabet Σ and a symbol $c \notin \Sigma$, If $L = X\,c\,Y,\ X, Y \subset \Sigma^*$ is PI-LIG language, then X or Y is a PI-PDA language.*

Lemma 2. *Any left-right-coded palindrome and its reversal are not PI-PDA languages.*

Proof (of Proposition 4). We prove this proposition by showing the following two facts.

1. The copy language is in $\mathcal{C}$(PI-LIG), but not in $\mathcal{C}$(PDA).
2. Let L_x be the set of the left-right-coded palindromes. The language $L = L_x c L_x$ is in $\mathcal{C}$(PDA) but not in $\mathcal{C}$(PI-LIG), where c is an input symbol that does not occur in L_x.

As the first fact has been shown in Example 1, and the set $L_x c L_x$ is a CFL, the remainder of the proof is to show that L is not a PI-LIG language. Suppose that $L \in \mathcal{C}$(PI-LIG), then by Lemma 1 L_x must be a PI-PDA language. However, by Lemma 2 this cannot be, hence L is not a PI-LIG language. □

3.3 Push-Input PDAs with ϵ- Transition Rule

By Proposition 3, there is no PI-PDA for the binary left-coded palindrome. On the other hand, we can construct a push-input PDA with ϵ-transition pop-up rules for this language. Fig. 3 shows a state transition diagram for a push-input-PDA with ϵ-transition pop-up rules, which is transformed from PDA M_{BCP} in Example 3. The problem whether all PDAs, or LIGs, can be transformed into push-input PDA, or LIGs respectively, with ϵ-transition rules is currently open.

4 Bottom-Up Parsing for LIGs

PDAs and LIGs are intrinsically top-down parsing procedures. These direct procedures are generally nondeterministic, and the parsing may require exponential time. A bottom-up parsing method for LIGs shown in this section eliminates stack symbols, thus saving computational cost. This method was implemented in Prolog and tested for several LIGs.

Fig. 4 shows forward inference rues for bottom-up parsing. In parsing an input string $a_1 a_2 \cdots a_n$, any intermediate results are saved in terms of the form

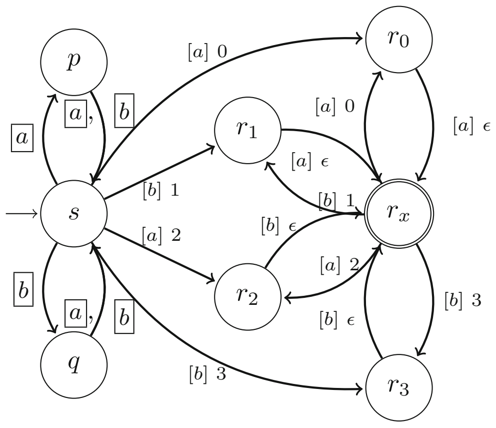


Fig. 3. State Transition Diagram of the Push-Input PDA with ϵ-Transition Pop-up Rules for Binary Coded Palindromes Transformed from M_{BCP} in Fig. 2

$d((i,i'),(j,j'),p-q)$, indicating that q is a combinational state of p and that the transition from p to q derives a substring of $a_i \cdots a_{i'}$ and $a_{j'} \cdots a_j$ in the reverse order of the input string. Only the rules 1, 3, 6 and 9 are sufficient for parsing PDAs. The overlines indicates that input symbols are used in the backward-input rules.

4.1 Example 4: Set $\{a^n b^n c^n \mid n \geq 1\}$

The following four parsing rules are transformed from the LIG in Section 2.

1. $\overline{c_j} \to d((i,i'),(j-1,j),p-s)$, For rule $p \to s\,c$.
2. $a_i, b_{i'+1}, d((i,i'),(j,j'),p-s) \to d((i-1,i'+1),(j,j')), s-q)$, for rules $s \to a\,p\,[a],\ s\,[a] \to b\,q$.
3. $a_i, b_{i'+1}, d((i,i'),(j,j'),p-s) \to d((i-1,i'+1),(j,j')), p-q)$, for rules $s \to a\,p\,[a],\ q\,[a] \to b\,q$.
4. $d((i,i'),(j,j'),p-s), d((i',i''),(j',j''),s-q) \to ((i,i''),(j,j''),p-q)$.

The following sequence of terms is an example of bottom-up derivation for string *aabbcc*.

$$\begin{aligned}
&\{a_1, a_2, b_3, b_4, c_5, c_6\}\\
&\Rightarrow \{a_1, a_2, b_3, b_4, d((x,x'),(1,2),p-s), d((y,y'),(0,1),p-s)\}\\
&\Rightarrow \{a_1 d((1,3),(1,2),s-q), d((y,y'),(0,1),p-s), b_4,\}\\
&\Rightarrow \{a_1, d((1,3),(0,2),p-q), b_4,\}\\
&\Rightarrow \{d((0,4),(0,2),s-q)\}.
\end{aligned}$$

The derivation tree in Fig. 5 represents the relation of this parsing, which is based on the normalized LIG for $\{a^n b^n c^n \mid n \geq 1\}$.

1. $a_i \to d((i-1,i),(y,y'),p-q)$ for each rule $p \to a_i\, q$ and for any y and y'.
2. $\overline{a_j} \to d((x,x),(j-1,j),p-q)$ for each rule $p \to q\, a_j$ and for any x and x'.
3. $a_i, a_{i+1} \to d((i-1,i+1),(y,y'),p-r)$
   for rules $p \to a_i\, q[c]$ and $q[c] \to a_{i+1}\, r$ and for any y and y'.
4. $a_i, \overline{a_j} \to d((i-1,i),(j-1,j),p-r)$ for rules $p \to a_i\, q[c]$ and $q[c] \to r\, a_j$.
5. $\overline{a_j}, \overline{a_{j+1}} \to d((x,x'),(j-1,j+1),p-r)$ for rules $p \to q[c]\, a_{j+1}$ and $q[c] \to r\, a_j$.
6. $a_i, a_{i'+1}, d((i,i'),(j,j'),q-q') \to d((i-1,i'+1),(j,j'),p-r)$
   for rules $p \to a_i\, q[c]$ and $q'[c] \to a_{i'+1} r$ in G.
7. $a_i, \overline{a_j}, d((i,i'),(j,j'),q-q') \to d((i-1,i'),(j-1,j'),p-r)$
   for rules $p \to a_i\, q[c]$ and $q'[c] \to r\, a_j$ in G.
8. $\overline{a_j}, \overline{a_{j'+1}}, d((i,i'),(j,j'),q-q') \to d((i,i'),(j-1,j'+1),p-r)$
   for rules $p \to q[c]\, a_i$ and $q'[c] \to r\, a_j$ in G.
9. $d((i,i'),(j,j'),p-q), d((i',i''),(j',j''),q-r) \to ((i,i''),(j,j''),p-r)$
   for any states p,q,r and for any $i,j,i',j,' i'',j''$.

Fig. 4. Forward Inference Rules for Bottom-up Parsing for LIGs (Only rules 1, 3, 6 and 9 are used for PDAs)

4.2 Computation Time of Bottom-Up Parsing

Let N_w be the number of the terms generated in the bottom-up parsing for a string w. The computation time for parsing w is estimated by $N_w \cdot K_w$, where K_w is the maximum number of steps required to generate a term from two terms by Rule 9. The factor K_w depends on the indexing to select the terms, and the maximum is not greater than $|w|$ for PDAs and $|w|^2$ for LIGs. Without any restriction on N_w and K_w, the computation time required for the parsing PDAs is $O(n^2 \cdot n) = O(n^3)$, where n is the length of input strings. The computation time for parsing LIGs is $O(n^4 \cdot n^2) = O(n^6)$, which is equal to the order of the previously known upper bound for LIGs and TAGs [2,8]. However, the factor N_w is much less than $O(n^2)$ for PDAs and $O(n^4)$ for LIGs in many cases: the factor is not greater than $O(n^2)$ for the LIGs shown in the next section.

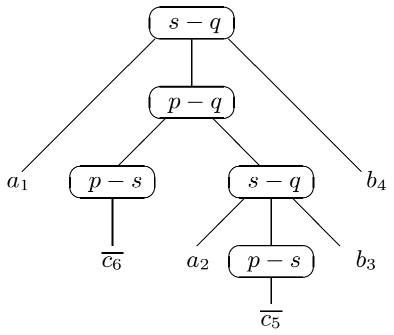


Fig. 5. A Derivation Tree for $aabbcc$ and the Normalized LIG for $\{a^n b^n c^n \mid n \geq 1\}$

As this bottom-up parsing is a process of forward inference by the parsing rules, we can use any efficient reasoning method for production systems such as the Rete algorithm [4] in the parsing.

5 Synthesized PI-PDAs and PI-LIGs

PI-PDAs and PI-LIGs for several fundamental formal languages are synthesized by a grammatical inference system LIG Learner written in Prolog as the first step for learning LIGs. The system is composed of a top-down parser, rule generation and search for rule sets. The top-down parser is used to pre-process the rule generation and to check any generated rule set against negative samples. The number of rules in each synthesized grammar is minimized, as the system searches for the rule sets by using iterative deepening. More details of the system with earlier results are described in [10].

The LIG Learner synthesized PI-PDAs and PI-LIGs in the following list. The LIGs (e) and (f) have been shown as examples in Section 2.1.

(a) the balanced parenthesis language: $s\,[a] \to b\,s,\; s \to a\,s\,[a]$

(b) the set of palindromes over $\{a, b\}$:

$$s \to a\,s\,[a],\; s \to b\,s\,[b],\; s \to a\,p,\; s \to b\,p,$$
$$s\,[a] \to a\,p,\; s\,[b] \to b\,p,\; p\,[a] \to a\,p,\; p\,[b] \to b\,p.$$

(c) the set of strings with same number of a's and b's, $\{w \mid \#_a(w) = \#_b(w)\}$:

$$s\,[b] \to a\,s,\; s \to b\,s\,[b],\; s\,[a] \to b\,s,\; s \to a\,s\,[a].$$

(d) the set of strings with twice as many a's as b's, $\{w \mid \#_a(w) = 2 \cdot \#_b(w)\}$:

$$s \to a\,p\,[a],\; s \to b\,s[b],\; s\,[a] \to b\,s,\; s\,[b] \to a\,p,$$
$$p \to b\,p[b],\; p \to a\,s,\; p\,[a] \to b\,p.$$

(e) copy language, $\{ww \mid w \in \{a, b\}\}$: the set of strings with the form ww.

$$p\,[b] \to p\,b,\; p\,[a] \to p\,a,\; s\,[b] \to p\,b,\; s \to b\,s\,[a],$$
$$s\,[a] \to p\,a,\; s \to a\,s\,[a].$$

(f) the set $\{a^n b^n c^n \mid 1 \le n\}$: $q\,[a] \to b\,q,\; s\,[a] \to b\,q,\; p \to s\,c,\; s \to a\,p\,[a].$

(g) the set $\{a^n b^n c^m \mid 0 \le m \le n\}$:

$$q\,[a] \to b\,q,\; p \to a\,p[a],\; s\,[a] \to b\,q,\; p \to s\,c,\; s \to a\,p\,[a].$$

(h) the set $\{a^i b^j c^i d^j \mid 1 \le i, j\}$:

$$s \to a\,p\,[a],\; s \to a\,s\,[a],\; p \to b\,p\,[b],\; p\,[b] \to q\,d,\; q\,[a] \to c\,q,\; q\,[b] \to q\,d.$$

The rules in (a) - (d) are PI-PDAs for CFLs, whereas the languages (e) – (h) are non-context-free. The language (h) is related to pseudo-knots in RNA secondary structure [11], where some basic pairs occur in crossed fashion. All the states are final states in the LIGs except PDA (d), where s is the only final state.

The experimental results were obtained using an AMD Athlon(tm) 64 X2 Dual Core processor with a 2.2 GHz clock and SWI-Prolog for Windows. The positive samples in the experiment are ordered strings within the length of eight or nine, which are generated by the programs. Only the first parts of the strings, generally less than 10 to 15 samples, are used for synthesizing the rules, and the remainder of the strings is used for checking the correctness of the rule sets. The system derives strings within the length of eight or nine from each generated rule set for the consistency check. We also checked the correctness of the synthesized grammars by enumerating the numbers of strings.

The computation time for learning each of PDAs (a), (b), (c) and the LIG (f) was less than 0.1 second and the time for each of the other grammars is less than 3 seconds. Comparing with learning CFGs in Synapse system [9], the number of rules in each PDA is 50-80 % less than that of an equivalent CFG in revised Chomsky normal form, and the computational time of the PDA is generally less than that of the CFG.

6 Concluding Remarks

In this paper, we discussed LIGs including PDAs, the push-input form and bottom-up parsing of these grammars. The results are summarized as follows:

– The class of PI-LIG languages without ϵ-transition rule, $\mathcal{C}$(PI-PDA), is incomparable with the class of PDA languages (or CFLs).
– A simple bottom-up parsing method for LIGs is presented, in which the stack symbols are eliminated at the first step of the parsing. This method is simpler than the other parsing methods for LIGs previously published.
– In spite of the restriction of the push-input form, $\mathcal{C}$(PI-PDA) and $\mathcal{C}$(PI-LIG) includes fundamental context-free and context-sensitive languages, which are synthesized by a grammatical inference system LIG Learner.

Based on the results in this paper, our current work is focused on incremental learning of LIGs using bridging rule generation [9], which is based on incomplete trees generated by the bottom-up parsing of positive samples. The experimental results of first-step machine learning of PI-LIGs in Section 5 suggest that learning PDAs in push-input form is more efficient than learning CFGs, and that machine learning of LIGs has potentiality of synthesizing more complex and powerful grammars.

The other subjects and problems for future research include:

– Clarification of the computational complexity of the bottom-up parsing and improvement of the practical procedure.
– More detailed characterization of push-input PDA and LIG languages. Are the classes of the languages push-input PDAs and LIGs with ϵ-transition pop-up rules equal to $\mathcal{C}$(PDA) and $\mathcal{C}$(LIG), respectively ?

Acknowledgments. The authors would like to thank Yusuke Natsume for his help in testing the LIG parser and preparing the manuscript.

References

1. Aho, A.V.: Indexed grammars: An extension of context-free grammars. J. ACM 15(4), 647–671 (1968)
2. Alonso, M., de la Clergerie, E., Diaz, V., Vilares, M.: Relating tabular parsing algorithms for LIG and TAG. New Developments in Parsing Technology 23, 157–184 (2005)
3. Duske, J., Parchmann, R.: Linear indexed languages. Theoretical Computer Science 32(1), 47–60 (1984)
4. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1), 17–37 (1982)
5. Goldstine, J., Price, J.K., Wotschke, D.: On reducing the number of stack symbols in a PDA. Theory of Computing Systems 26(4), 313–326 (1993)
6. Joshi, A., Levy, L., Takahashi, M.: Tree adjunct grammars. Journal of Computer and System Sciences 10(1), 136–163 (1975)
7. Joshi, A., Shanker, K., Weir, D.: The convergence of mildly context-sensitive grammar formalisms. In: Foundational Issues in Natural Language Processing, pp. 31–81. MIT Press (1991)
8. Kallmeyer, L.: Parsing Beyond Context-Free Grammars. Springer (2005)
9. Nakamura, K.: Incremental Learning of Context Free Grammars by Bridging Rule Generation and Search for Semi-optimum Rule Sets. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 72–83. Springer, Heidelberg (2006)
10. Nakamura, K., Imada, K.: Towards incremental learning of mildly context-sensitive grammars. In: 10th International Conference on Machine Learning and Applications (ICMLA 2011), vol. 1, pp. 223–228. IEEE (2011)
11. Rivas, E., Eddy, S.R.: The language of RNA: a formal grammar that includes pseudoknots. Bioinformatics 16, 334–340 (2000)

Asynchronous PC Systems of Pushdown Automata

Friedrich Otto

Fachbereich Elektrotechnik/Informatik
Universität Kassel
34109 Kassel, Germany
otto@theory.informatik.uni-kassel.de

Abstract. We introduce *asynchronous* variants of the PC systems of pushdown automata of Csuhaj-Varjú et. al. These are obtained by using a *response symbol* in addition to the usual *query symbols*. Our main result states that centralized asynchronous PC systems of pushdown automata of degree n that work in returning mode have exactly the same expressive power as n-head pushdown automata. This holds in the nondeterministic as well as in the deterministic case.

Keywords: PC system of pushdown automata, centralized PC system, asynchronous PC system, multi-head pushdown automaton.

1 Introduction

Parallel communicating grammar systems, or *PC grammar systems* for short, have been invented to realize the so-called *class room model* of cooperation [5]. Here a group of experts, modelled by grammars, work together in order to produce a document, that is, an output word. These experts work on their own, but synchronously, and they exchange information on request.

In the literature many different types and variants of PC grammar systems have been studied (see, e.g., [5,7]). The notion of PC system has also been carried over to various types of automata. Here we are interested in the *PC systems of pushdown automata* introduced in [6], which we will modify into *asynchronous PC systems of pushdown automata*. In a PC system of pushdown automata, a finite number n of pushdown automata, say $A_1, \ldots, A_n$, work in parallel in a synchronous way, where the number n of components is called the *degree* of the PC system. If one of these pushdown automata, say A_i, encounters a special *query symbol* as the topmost symbol on its pushdown store, say K_j, then a *communication step* takes place: the symbol K_j on the top of the pushdown of A_i is replaced by the complete pushdown contents of the pushdown automaton A_j, provided that the topmost symbol of it is not a query symbol. The PC system is said to work in *returning mode*, if by this communication step the contents of the pushdown of A_j is reset to its initial symbol Z_j.

If there is only one component, called the *master* of the system, that can use query symbols, then the PC system is called *centralized*. It is known that a centralized PC system of pushdown automata of degree n that is working in returning mode can simulate an n-head pushdown automaton [6]. In [1], it was claimed that also conversely, centralized PC systems of pushdown automata of degree n working in returning mode can be simulated by n-head pushdown automata, but it was shown in [8] that the proof given in [1] is incorrect. In fact, it was established by Petersen that every recursively enumerable language is accepted by a centralized PC system of pushdown automata of degree two that is working in returning mode [9]. It follows that multi-head pushdown automata are strictly weaker than centralized PC systems of pushdown automata that work in returning mode.

As observed in [8] it is the inherent synchronization of the various components of a PC system of pushdown automata that makes these systems so powerful. This is further exemplified in [9], as the proofs given in that paper make use of this synchronous behaviour in essential ways. Here we do away with this synchronous behaviour by defining *asynchronous PC systems of pushdown automata*, abbreviated as APCPDA. These systems are obtained by introducing an additional *response symbol*. Assume that $\mathcal{M}$ is an APCPDA of degree n with components $A_1, \ldots, A_n$. If one of these pushdown automata, say A_i, encounters a special *query symbol* as the topmost symbol on its pushdown store, say K_j, then it wishes to perform a communication (see above). However, this is possible only if the pushdown automaton A_j has the special *response symbol* R as the topmost symbol on its pushdown. If this is the case, then the symbol K_j on the top of the pushdown of A_i is replaced by the pushdown contents of A_j without the response symbol R, and the contents of the pushdown of A_j is reset to its initial symbol Z_j; otherwise, A_i will have to wait until the communication is enabled. Symmetrically, if A_j has the special response symbol R as the topmost symbol on its pushdown, then it has to wait until A_i is ready for the corresponding communication. Thus, in this model the component, the pushdown contents of which is sent to a requesting component, is fully aware of this fact and of the time of this communication. Further, as the sending (the receiving) component has to wait until the receiving (the sending) component is ready for the communication, we see that the various components do not work in complete synchronization anymore. That's why we choose to call this model *asynchronous*. As our main result we will show that the centralized asynchronous PC systems of pushdown automata of degree n that work in returning mode correspond in expressive power exactly to the multi-head pushdown automata of degree n. This result also holds for the deterministic case.

The paper is structured as follows. In Section 2 we restate the definition of PC systems of pushdown automata from [6] in short and recall some of their properties, and we define the asynchronous PC systems of pushdown automata. We also present a detailed example, and we prove that asynchronous PC systems of pushdown automata working in returning mode can be simulated by PC systems of the same type and degree working in nonreturning mode. In the next section we recall the definition of the multi-head pushdown automaton and derive our main results. Then in Section 4, we discuss centralized asynchronous

PC systems of pushdown automata working in nonreturning mode. The paper closes with a short summary and some open problems.

2 PC Systems of Pushdown Automata

First we restate the definition of the PC system of pushdown automata from [6].

Definition 1. *A* PC system of pushdown automata *is given through a tuple* $\mathcal{A} = (\Sigma, \Gamma, A_1, \ldots, A_n, K)$, *where*

- Σ *is a finite input alphabet and* Γ *is a finite pushdown alphabet,*
- *for each* $1 \leq i \leq n$, $A_i = (Q_i, \Sigma, \Gamma, ¢, \delta_i, q_i, Z_i, F_i)$ *is a nondeterministic pushdown automaton with finite set of internal states* Q_i, *initial state* $q_i \in Q_i$ *and set of final states* $F_i \subseteq Q_i$, *input alphabet* Σ, *pushdown alphabet* Γ, *end marker* $¢ \notin \Sigma$, *initial pushdown symbol* $Z_i \in \Gamma$, *and transition relation* $\delta_i : Q_i \times (\Sigma \cup \{¢, \varepsilon\}) \times \Gamma \to 2^{Q_i \times \Gamma^*}$,
- *and* $K \subseteq \{K_1, K_2, \ldots, K_n\} \subseteq \Gamma$ *is a set of* query symbols.

Here the pushdown automata $A_1, \ldots, A_n$ are the *components* of the system $\mathcal{A}$, and the integer n is called the *degree* of this PC system.

Definition 1 (cont.). *A* configuration *of* $\mathcal{A}$ *is described by a* $3n$*-tuple*

$$(s_1, x_1¢, \alpha_1, s_2, x_2¢, \alpha_2, \ldots, s_n, x_n¢, \alpha_n),$$

where, for $1 \leq i \leq n$,

- $s_i \in Q_i$ *is the current state of component* A_i,
- $x_i \in \Sigma^*$ *is the remaining part of the input which has not yet been read by component* A_i, *and*
- $\alpha_i \in \Gamma^*$ *is the current contents of the pushdown of* A_i, *where the first symbol of* α_i *is the topmost symbol on the pushdown.*

On the set of configurations $\mathcal{A}$ induces a *computation relation* $\vdash^*_{\mathcal{A},r}$ that is the reflexive and transitive closure of the following relation $\vdash_{\mathcal{A},r}$.

Definition 2. *For configurations* $(s_1, x_1¢, c_1\alpha_1, \ldots, s_n, x_n¢, c_n\alpha_n)$ *and* $(p_1, y_1¢, \beta_1, \ldots, p_n, y_n¢, \beta_n)$, *where* $c_1, \ldots, c_n \in \Gamma$,

$$(s_1, x_1¢, c_1\alpha_1, \ldots, s_n, x_n¢, c_n\alpha_n) \vdash_{\mathcal{A},r} (p_1, y_1¢, \beta_1, \ldots, p_n, y_n¢, \beta_n)$$

if and only if one of the following two conditions is satisfied:

(1) $K \cap \{c_1, \ldots, c_n\} = \emptyset$, *and for all* $1 \leq i \leq n$, $x_i = a_i y_i$ *for some* $a_i \in \Sigma \cup \{\varepsilon\}$, $(p_i, \gamma_i) \in \delta_i(s_i, a_i, c_i)$, *and* $\beta_i = \gamma_i\alpha_i$, *or* $x_i = \varepsilon = y_i$, $(p_i, \gamma_i) \in \delta_i(s_i, ¢, c_i)$, *and* $\beta_i = \gamma_i\alpha_i$, *or*

(2) – $K \cap \{c_1, \ldots, c_n\} \neq \emptyset$,
  – *for all* $i \in \{1, \ldots, n\}$ *such that* $c_i = K_{j_i}$ *and* $c_{j_i} \notin K$, $\beta_i = c_{j_i}\alpha_{j_i}\alpha_i$ *and* $\beta_{j_i} = Z_{j_i}$,

– *$\beta_r = c_r\alpha_r$ for all other values of $r \in \{1,\ldots,n\}$,*
– *$y_t = x_t$ and $p_t = s_t$ for all $t \in \{1,\ldots,n\}$.*

The steps of form (1) are called *local steps*, as in them each component A_i ($1 \leq i \leq n$) performs a local step, concurrently, but otherwise independently. The steps of form (2) are called *communication steps*, as in them the topmost symbol K_{j_i} on the pushdown of component A_i is replaced by the complete contents $c_{j_i}\alpha_{j_i}$ of the pushdown of component A_{j_i}, provided that the topmost symbol c_{j_i} is itself not a query symbol. At this time also the pushdown of A_{j_i} is reset to its initial symbol Z_{j_i}. Accordingly, it is said that $\mathcal{A}$ works in *returning mode.* If the contents of the pushdown of A_{j_i} were to remain unchanged by the communication step, then $\mathcal{A}$ would work in *nonreturning mode.* The computation relation for this mode is denoted by $\vdash_{\mathcal{A}}^*$.

Definition 3. (a) *The* language $L_r(\mathcal{A})$ *that is accepted by $\mathcal{A}$ working in returning mode is defined by*

$$L_r(\mathcal{A}) = \{\, w \in \Sigma^* \mid (q_1, w¢, Z_1, \ldots, q_n, w¢, Z_n) \vdash_{\mathcal{A},r}^* (s_1, ¢, \alpha_1, \ldots, s_n, ¢, \alpha_n) \text{ for some } s_i \in F_i \text{ and } \alpha_i \in \Gamma^*,\ 1 \leq i \leq n \,\},$$

and the language $L(\mathcal{A})$ that is accepted by $\mathcal{A}$ working in nonreturning mode is defined by

$$L(\mathcal{A}) = \{\, w \in \Sigma^* \mid (q_1, w¢, Z_1, \ldots, q_n, w¢, Z_n) \vdash_{\mathcal{A}}^* (s_1, ¢, \alpha_1, \ldots, s_n, ¢, \alpha_n) \text{ for some } s_i \in F_i \text{ and } \alpha_i \in \Gamma^*,\ 1 \leq i \leq n \,\}.$$

(b) *By $\mathcal{L}_r(\mathsf{PCPDA}(n))$ we denote the class of languages that are accepted by PC systems of pushdown automata of degree n working in returning mode, and by $\mathcal{L}(\mathsf{PCPDA}(n))$ we denote the class of languages that are accepted by PC systems of pushdown automata of degree n working in nonreturning mode.*

(c) *A PC system of pushdown automata $\mathcal{A} = (\Sigma, \Gamma, A_1, \ldots, A_n, K)$ is* centralized *if there is only a single component, say A_1, that can use query symbols. In this case, A_1 is called the* master *of the system $\mathcal{A}$. By $\mathcal{L}_r(\mathsf{CPCPDA}(n))$ we denote the class of languages that are accepted by centralized PC systems of pushdown automata of degree n working in returning mode, and by $\mathcal{L}(\mathsf{CPCPDA}(n))$ we denote the class of languages that are accepted by centralized PC systems of pushdown automata of degree n working in nonreturning mode.*

The following result is known on the expressive power of PC systems of pushdown automata.

Theorem 4. [6] *The classes $\mathcal{L}(\mathsf{PCPDA}(2))$ and $\mathcal{L}_r(\mathsf{PCPDA}(3))$ coincide with the class of all recursively enumerable languages.*

Recently the result on PC systems of pushdown automata working in returning mode has been improved as follows.

Theorem 5. [9] *The class $\mathcal{L}_r(\mathsf{CPCPDA}(2))$ coincides with the class of all recursively enumerable languages.*

Let $\mathcal{A} = (\Sigma, \Gamma, A_1, \ldots, A_n, K)$ be a centralized PC system of pushdown automata, and let

$$(s_1, x_1 ¢, K_j \alpha_1, \ldots, s_j, x_j ¢, \alpha_j, \ldots) \vdash_{\mathcal{A},r} (s_1, x_1 ¢, \alpha_j \alpha_1, \ldots, s_j, x_j ¢, Z_j, \ldots)$$

be a communication step of $\mathcal{A}$. The component A_1 is *actively* involved in this communication step, while component A_j is only *passively* involved in it, that is, it does not really know about its involvement in this step. It can at best realize its involvement *after* the communication has taken place. On the other hand, A_1 knows exactly how many local steps A_j has executed before the communication step, as A_1 and A_j perform their local steps strictly synchronously. We now define a new variant of PC systems of pushdown automata, in which we use a *response symbol* in addition to the query symbols. In this way we will

– enable the component A_j to become an active participant in the communication above, and
– break the strict synchronicity of local steps between the various components of a PC system.

Definition 6. *An* asynchronous PC system of pushdown automata *is given through a tuple $\mathcal{A} = (\Sigma, \Gamma, A_1, \ldots, A_n, K, R)$, where*

– *Σ, Γ, $A_1, \ldots, A_n$, and K are defined as in Definition 1, and*
– *$R \in \Gamma \smallsetminus K$ is a special* response symbol *such that $R \neq Z_i$ for all $i = 1, \ldots, n$.*

Configurations of these systems are defined in the same way as for PC systems of pushdown automata. On the set of configurations $\mathcal{A}$ induces a *computation relation* $\vdash^*_{\mathcal{A},r}$ that is the reflexive and transitive closure of the following relation $\vdash_{\mathcal{A},r}$.

Definition 7. *For configurations $(s_1, x_1 ¢, c_1 \alpha_1, \ldots, s_n, x_n ¢, c_n \alpha_n)$ and $(p_1, y_1 ¢, \beta_1, \ldots, p_n, y_n ¢, \beta_n)$, where $c_1, \ldots, c_n \in \Gamma$,*

$$(s_1, x_1 ¢, c_1 \alpha_1, \ldots, s_n, x_n ¢, c_n \alpha_n) \vdash_{\mathcal{A},r} (p_1, y_1 ¢, \beta_1, \ldots, p_n, y_n ¢, \beta_n)$$

if and only if one of the following two conditions is satisfied:

(1) *if there are indices $i, j \in \{1, \ldots, n\}$ such that $c_i = K_j$ and $c_j = R$, then a* communication step *takes place:*
   – *for all $i \in \{1, \ldots, n\}$ and all $j \in \{1, \ldots, n\}$ such that $c_i = K_j$ and $c_j = R$, $\beta_i = \alpha_j \alpha_i$ and $\beta_j = Z_j$,*
   – *$\beta_r = c_r \alpha_r$ for all other values of $r \in \{1, \ldots, n\}$, and*
   – *$y_t = x_t$ and $p_t = s_t$ for all $t \in \{1, \ldots, n\}$, or*

(2) *if there are no such indices i and j, then a* local step *takes place:*

- *for all* $i \in \{1, \ldots, n\}$ *such that* $c_i \in K \cup \{R\}$, $\beta_i = c_i\alpha_i$, $y_i = x_i$ *and* $p_i = s_i$,
- *for all other values of* $i \in \{1, \ldots, n\}$, $x_i = a_i y_i$ *for some* $a_i \in \Sigma \cup \{\varepsilon\}$, $(p_i, \gamma_i) \in \delta_i(s_i, a_i, c_i)$, *and* $\beta_i = \gamma_i\alpha_i$, *or* $x_i = \varepsilon = y_i$, $(p_i, \gamma_i) \in \delta_i(s_i, ¢, c_i)$, *and* $\beta_i = \gamma_i\alpha_i$.

Observe that a *communication step* is executed as soon as there are two components, say A_i and A_j, such that the topmost symbol on the pushdown of A_i is the query symbol K_j, and the topmost symbol on the pushdown of A_j is the response symbol R. In this case, the symbol K_j is replaced by the pushdown contents of A_j without the symbol R, and the pushdown contents of A_j is reset to its initial symbol Z_j. In fact, such communications are carried out for all components that satisfy the above requirements. If no communication is possible, then all components that have neither a query symbol nor a response symbol on the top of their pushdowns execute a single step of a local computation, while all those components that have a query symbol or a response symbol at the top of their pushdowns just wait. As in a communication between A_i and A_j as above, the pushdown contents of A_j is reset to the initial symbol Z_j, we say that the above definition describes the *returning mode* of operation for $\mathcal{A}$. If we require that in the above communication, just the response symbol R is deleted from the pushdown of A_j, then we say that $\mathcal{A}$ works in the *nonreturning mode*, which is denoted by $\vdash_{\mathcal{A}}^*$.

The *language* $L_r(\mathcal{A})$ that is accepted by $\mathcal{A}$ working in returning mode and the language $L(\mathcal{A})$ that is accepted by $\mathcal{A}$ working in nonreturning mode are defined as in Definition 3. By $\mathcal{L}_r(\mathsf{APCPDA}(n))$ we denote the class of languages that are accepted by asynchronous PC systems of pushdown automata of degree n working in returning mode, and by $\mathcal{L}(\mathsf{APCPDA}(n))$ we denote the class of languages that are accepted by asynchronous PC systems of pushdown automata of degree n working in nonreturning mode. An asynchronous PC system of pushdown automata $\mathcal{A} = (\Sigma, \Gamma, A_1, \ldots, A_n, K, R)$ is called *centralized* if there is only a single component, say A_1, that can use query symbols. By $\mathcal{L}_r(\mathsf{CAPCPDA}(n))$ we denote the class of languages that are accepted by centralized asynchronous PC systems of pushdown automata of degree n working in returning mode, and by $\mathcal{L}(\mathsf{CAPCPDA}(n))$ we denote the class of languages that are accepted by centralized asynchronous PC systems of pushdown automata of degree n working in nonreturning mode. Finally, an asynchronous PC system of pushdown automata $\mathcal{A} = (\Sigma, \Gamma, A_1, \ldots, A_n, K, R)$ is called *deterministic*, if all its components are deterministic pushdown automata. By $\mathcal{L}_r(\mathsf{APCDPDA}(n))$ we denote the class of languages that are accepted by asynchronous PC systems of deterministic pushdown automata of degree n working in returning mode, and by $\mathcal{L}(\mathsf{APCDPDA}(n))$ we denote the class of languages that are accepted by asynchronous PC systems of deterministic pushdown automata of degree n working in nonreturning mode, and analogously for centralized systems.

Next we present a simple example of an asynchronous PC system of pushdown automata.

Example 8. Let $\mathcal{A} = (\Sigma, \Gamma, A_1, A_2, \{K_2\}, R)$ be the centralized asynchronous PC system of degree 2 that contains the components $A_1 = (Q_1, \Sigma, \Gamma, ¢, \delta_1, p_1, Z_1, F_1)$ and $A_2 = (Q_2, \Sigma, \Gamma, ¢, \delta_2, q_1, Z_2, F_2)$, where

- $Q_1 = \{p_1, p_2, p_3, p_4\}$ and $Q_2 = \{q_1, q_2, q_3\}$,
- $F_1 = \{p_4\}$ and $F_2 = \{q_3\}$,
- $\Sigma = \{a, b, c\}$ and $\Gamma = \{Z_1, Z_2, A, B, C, K_2, R\}$, and
- the transition relations are defined as follows:

$$\begin{array}{llll}
(1) & \delta_1(p_1, a, Z_1) = \{(p_1, Z_1)\}, & (9) & \delta_2(q_1, a, Z_2) = \{(q_1, RAZ_2)\}, \\
(2) & \delta_1(p_1, b, Z_1) = \{(p_1, Z_1)\}, & (10) & \delta_2(q_1, b, Z_2) = \{(q_1, RBZ_2)\}, \\
(3) & \delta_1(p_1, c, Z_1) = \{(p_2, K_2Z_1)\}, & (11) & \delta_2(q_1, c, Z_2) = \{(q_2, RCZ_2)\}, \\
(4) & \delta_1(p_2, a, A) \ = \{(p_3, \varepsilon)\}, & (12) & \delta_2(q_2, a, Z_2) = \{(q_2, Z_2)\}, \\
(5) & \delta_1(p_2, b, B) \ = \{(p_3, \varepsilon)\}, & (13) & \delta_2(q_2, b, Z_2) = \{(q_2, Z_2)\}, \\
(6) & \delta_1(p_2, ¢, C) \ = \{(p_4, \varepsilon)\}, & (14) & \delta_2(q_2, ¢, Z_2) = \{(q_3, Z_2)\}, \\
(7) & \delta_1(p_3, \varepsilon, Z_2) = \{(p_2, K_2)\}, & (15) & \delta_2(q_3, ¢, Z_2) = \{(q_3, Z_2)\}. \\
(8) & \delta_1(p_4, ¢, Z_2) = \{(p_4, Z_2)\}, & &
\end{array}$$

On input *abcab*, the system $\mathcal{A}$ executes the following computation:

$$\begin{array}{rl}
(p_1, abcab¢, Z_1, q_1, abcab¢, Z_2) & \vdash_{\mathcal{A},r} (p_1, bcab¢, Z_1, q_1, bcab¢, RAZ_2) \\
& \vdash_{\mathcal{A},r} (p_1, cab¢, Z_1, q_1, bcab¢, RAZ_2) \\
& \vdash_{\mathcal{A},r} (p_2, ab¢, K_2Z_1, q_1, bcab¢, RAZ_2) \\
& \vdash_{\mathcal{A},r} (p_2, ab¢, AZ_2Z_1, q_1, bcab¢, Z_2) \\
& \vdash_{\mathcal{A},r} (p_3, b¢, Z_2Z_1, q_1, cab¢, RBZ_2) \\
& \vdash_{\mathcal{A},r} (p_2, b¢, K_2Z_1, q_1, cab¢, RBZ_2) \\
& \vdash_{\mathcal{A},r} (p_2, b¢, BZ_2Z_1, q_1, cab¢, Z_2) \\
& \vdash_{\mathcal{A},r} (p_3, ¢, Z_2Z_1, q_2, ab¢, RCZ_2) \\
& \vdash_{\mathcal{A},r} (p_2, ¢, K_2Z_1, q_2, ab¢, RCZ_2) \\
& \vdash_{\mathcal{A},r} (p_2, ¢, CZ_2Z_1, q_2, ab¢, Z_2) \\
& \vdash_{\mathcal{A},r} (p_4, ¢, Z_2Z_1, q_2, b¢, Z_2) \\
& \vdash_{\mathcal{A},r} (p_4, ¢, Z_2Z_1, q_2, ¢, Z_2) \\
& \vdash_{\mathcal{A},r} (p_4, ¢, Z_2Z_1, q_3, ¢, Z_2),
\end{array}$$

that is, $abcab \in L_r(\mathcal{A})$. On the other hand, it is easily checked that $\mathcal{A}$ cannot accept on input w, if w is not of the form $w = ucu$ for any $u \in \{a, b\}^*$. It follows that $L_r(\mathcal{A})$ is the copy language $L_{\text{copy}} = \{\, ucu \mid u \in \{a, b\}^* \,\}$, which is not even a growing context-sensitive language (see, e.g., [2]). Observe that $\mathcal{A}$ is in fact a deterministic system.

In a computation of an asynchronous PC system of pushdown automata, each component realizes its involvement in a communication step. If the system works in nonreturning mode, then in a communication step in which the pushdown contents of A_j is copied to A_i, the response symbol R is removed from the top of the pushdown of the sending component A_j. Now by using additional internal states, the component A_j can be forced to reset its pushdown to its bottom marker Z_j before it continues with its computation. This gives the following result.

Theorem 9. *Let $n \geq 2$, and let $\mathcal{A} \in \mathsf{APCPDA}(n)$. Then one can effectively construct a system $\mathcal{A}' \in \mathsf{APCPDA}(n)$ such that $L(\mathcal{A}') = L_r(\mathcal{A})$. In addition, if $\mathcal{A}$ is centralized and/or deterministic, then so is $\mathcal{A}'$.*

Thus, we have the following inclusions.

Corollary 10. *For all $n \geq 2$,* (a) $\mathcal{L}_r(\mathsf{CAPCDPDA}(n)) \subseteq \mathcal{L}(\mathsf{CAPCDPDA}(n))$.
(b) $\mathcal{L}_r(\mathsf{CAPCPDA}(n)) \subseteq \mathcal{L}(\mathsf{CAPCPDA}(n))$.
(c) $\mathcal{L}_r(\mathsf{APCDPDA}(n)) \subseteq \mathcal{L}(\mathsf{APCDPDA}(n))$.
(d) $\mathcal{L}_r(\mathsf{APCPDA}(n)) \subseteq \mathcal{L}(\mathsf{APCPDA}(n))$.

3 Multi-head Pushdown Automata

Next we repeat in short the definition of the multi-head pushdown automaton, where we follow the presentation in [6] (see also [3,4]).

Definition 11. *For $n \geq 1$, an n-*head pushdown automaton *is given through a 9-tuple $B = (n, Q, \Sigma, \Gamma, ¢, \delta, q_0, Z_0, F)$, where Q is a finite set of internal states, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is a set of final states, Σ is a finite input alphabet and Γ is a finite pushdown alphabet with initial pushdown symbol $Z_0 \in \Gamma$, the symbol $¢ \notin \Sigma$ is a special end marker, and $\delta : Q \times (\Sigma \cup \{¢, \varepsilon\})^n \times \Gamma \to 2^{Q \times \Gamma^*}$ is a transition relation. If $(q', \alpha) \in \delta(q, a_1, \ldots, a_n, X)$, then this means that B, when in state q with X as the topmost symbol on its pushdown and reading a_i with its i-th head $(1 \leq i \leq n)$, can change to state q' and replace the symbol X by the string α on the top of the pushdown. In addition, if $a_i \in \Sigma$, then head i moves one step to the right, and if $a_i = ¢$ or $a_i = \varepsilon$, then head i remains stationary.*

A configuration *of B is described by an $(n+2)$-tuple*

$$(q, x_1¢, \ldots, x_n¢, \alpha) \in Q \times (\Sigma^* \cdot \{¢\})^n \times \Gamma^*,$$

where q is the current internal state, x_i is the remaining part of the input still unread by head i $(1 \leq i \leq n)$, and α is the current contents of the pushdown, where the first symbol of α is the topmost symbol. By $\vdash_B$ we denote the single-step computation relation that B induces on its set of configurations, and $\vdash_B^*$ is its reflexive and transitive closure. If δ is a function $\delta : Q \times (\Sigma \cup \{¢, \varepsilon\})^n \times \Gamma \to Q \times \Gamma^*$ such that the induced transition relation on configurations of B is a function, then B is a deterministic *n-head pushdown automaton.*

The language $L(B)$ accepted by B is now defined as

$$L(B) = \{\, w \in \Sigma^* \mid (q_0, w¢, \ldots, w¢, Z_0) \vdash_B^* (s, ¢, \ldots, ¢, \alpha) \text{ for some } s \in F \text{ and } \alpha \in \Gamma^* \,\}.$$

In [6] it is shown that each language that is accepted by an n-head pushdown automaton is also accepted by some centralized PC system of pushdown automata of degree n that is working in returning mode. Here we carry this result over to asynchronous PC systems of pushdown automata.

Theorem 12. *If L is accepted by a (deterministic) n-head pushdown automaton, then it is also accepted by a centralized asynchronous PC system of (deterministic) pushdown automata of degree n that is working in returning mode.*

Proof. Let $B = (n, Q, \Sigma, \Gamma, ¢, \delta, q_0, Z_1, F)$ be an n-head (deterministic) pushdown automaton. If $n = 1$, then there is nothing to prove. So let $n \geq 2$.

We construct a centralized asynchronous PC system of (deterministic) pushdown automata $\mathcal{A} = (\Sigma, \Gamma_1, A_1, A_2, \ldots, A_n, K, R)$ of degree n that simulates B. This simulation will proceed as follows: first the master requests the symbols from all the other components that are currently under their input heads, and it stores this information within its finite-state control. Then based on this information, it can determine (deterministically) the next step of B that it must simulate. In this way we will be able to simulate B by a centralized asynchronous PC system of (deterministic) pushdown automata. Accordingly, the PC system $\mathcal{A}$ is defined as follows:

- $K = \{K_2, \ldots, K_n\}$ and $\Gamma_1 = \Gamma \cup K \cup \{R\} \cup \{Z_2, \ldots, Z_n\}$,
- $A_1 = (Q_1, \Sigma, \Gamma_1, ¢, \delta_1, q_0^{(1)}, Z_1, F_1)$ and
- $A_i = (Q_i, \Sigma, \Sigma \cup \{Z_i, R\}, ¢, \delta_i, q_0^{(i)}, Z_i, F_i)$, $i = 2, \ldots, n$, where
  - $Q_1 = \{q_0\} \cup \{\, q_{[p,\mu_1,\ldots,\mu_n]} \mid p \in Q, \mu_1, \ldots, \mu_n \in \Sigma \cup \{¢, \perp, \perp'\} \,\}$, and
  - $Q_i = \{q_0^{(i)}\}$ for $i = 2, \ldots, n$,
  - $q_0^{(1)} = q_0$, $F_1 = \{\, q_{[p,¢,\ldots,¢]} \mid p \in F \,\}$, and $F_i = \{q_0^{(i)}\}$, $i = 2, \ldots, n$,
  - the transition relation δ_1 is given by the following description, where $p \in Q$, $a_1, \ldots, a_n \in \Sigma \cup \{¢\}$, $A \in \Gamma$, $i \geq 2$, and $b_1, \ldots, b_n \in \Sigma \cup \{¢, \perp\}$:

$$\begin{array}{lrl}
(1) & \delta_1(q_0, a_1, Z_1) & = \{(q_{[q_0,a_1,\perp,\ldots,\perp]}, Z_1)\},\\
(2) & \delta_1(q_{[p,a_1,\ldots,a_{i-1},\perp,b_{i+1},\ldots]}, \varepsilon, A) & = \{(q_{[p,a_1,\ldots,a_{i-1},\perp',b_{i+1},\ldots]}, K_iA)\},\\
(3) & \delta_1(q_{[p,a_1,\ldots,a_{i-1},\perp',b_{i+1},\ldots]}, \varepsilon, a_i) & = \{(q_{[p,a_1,\ldots,a_{i-1},a_i,b_{i+1},\ldots]}, \varepsilon)\},\\
(4) & \delta_1(q_{[p,a_1,\ldots,a_n]}, \varepsilon, A) & = \{\, (q_{[p',b_1,\ldots,b_n]}, \alpha) \mid (p', \alpha) \in \delta(p, c_1, \ldots, c_n, A),\\
& & \quad c_j = a_j \text{ and } b_j = \perp, \text{ or } c_j = \varepsilon \text{ and } b_j = a_j, 1 \leq j \leq n \,\},
\end{array}$$

  - and the other transition relations are defined by

$$(5)\ \delta_i(q_0^{(i)}, a, Z_i) = \{(q_0^{(i)}, Ra)\} \text{ for all } a \in \Sigma \cup \{¢\},\ 2 \leq i \leq n.$$

In its finite-state control, A_1 remembers the actual state of B and the symbols that are currently under the heads of B. Based on this information and the symbol on the top of its pushdown, A_1 can determine the next step of B, which it then simulates (see (4)). In this step B may consume only some of the symbols $a_1, \ldots, a_n$, as some of its heads may just perform ε-steps. Accordingly, A_1 consumes only those symbols a_i that are consumed (read) by the corresponding heads of B. These symbols are replaced by the symbol $\perp$ within the finite-state control of A_1. Then using the rules (2) to (3), A_1 requests the next symbols these heads will read. After obtaining all the required symbols, A_1 can simulate the next step of B. It now follows easily that $L_r(\mathcal{A}) = L(B)$ holds, which completes the proof of Theorem 12. □

In fact, also the converse of Theorem 12 holds.

Theorem 13. *If L is accepted by a centralized asynchronous PC system of (deterministic) pushdown automata of degree n that is working in returning mode, then L is also accepted by some (deterministic) n-head pushdown automaton.*

Proof. Here we can follow the proof idea of Balan [1], which works correctly now that centralized PC systems of pushdown automata are considered that are asynchronous.

This simulation works as follows. Each of the n heads of the n-head pushdown automaton B will simulate the input head of one of the components of the centralized asynchronous PC system $\mathcal{A}$. First, using head 1, B simulates the master A_1 of the system $\mathcal{A}$ up to the point, where a query symbol K_j occurs as the topmost symbol on the pushdown. Then using head j, B simulates the component A_j using the pushdown of B. Thus, at the time of the communication step, the pushdown contents of B will correspond exactly to the pushdown contents of the master A_1 *after* execution of the communication step. Now for the PC systems of [6] the problem was that B cannot recognize this moment in time. However, as we consider an asynchronous PC system, the component A_j puts the response symbol R onto its pushdown, when it is ready for the communication to take place. Thus, at this moment B can switch back to simulating the master component A_1. Once the simulation of A_1 has been completed successfully, B can simulate the other components, one by one, as no more communication steps will occur. It follows that $L(B) = L_r(\mathcal{A})$ holds. $\square$

In summary we obtain the following characterization.

Corollary 14. *For all $n \geq 1$,*

$$\mathcal{L}_r(\mathsf{CAPCPDA}(n)) = \mathcal{L}(n\text{-}\mathsf{PDA}) \text{ and } \mathcal{L}_r(\mathsf{CAPCDPDA}(n)) = \mathcal{L}(n\text{-}\mathsf{DPDA}),$$

that is, a language is accepted by a centralized asynchronous PC system of (deterministic) pushdown automata of degree n that is working in returning mode if and only if it is accepted by a (deterministic) n-head pushdown automaton.

4 On Centralized Systems Working in Nonreturning Mode

We have seen in Corollary 10 that asynchronous PC systems of pushdown automata working in returning mode can be simulated by asynchronous PC systems of the same type and the same number of components working in nonreturning mode. However, in nonreturning mode, centralized APCPDA systems are actually much more expressive than in returning mode. Here the following result holds.

Theorem 15. *Each recursively enumerable language is accepted by a centralized asynchronous PC system of pushdown automata of degree two that is working in nonreturning mode, that is, $\mathcal{L}(\mathsf{CAPCPDA}(2)) = \mathsf{RE}$.*

It remains to determine the expressive power of centralized asynchronous PC systems of deterministic pushdown automata that work in nonreturning mode. We do not yet have a characterization for these systems, but we have the following result.

Theorem 16. *For all $n \geq 2$, $\mathcal{L}_r(\mathsf{CAPCDPDA}(n)) \subsetneq \mathcal{L}(\mathsf{CAPCDPDA}(n))$.*

Proof. It remains to show that the inclusions above are proper. For $n \geq 2$, let L_n denote the following language on $\Sigma = \{0, 1, \#, \$\}$:

$$L_n = \{\, w_1\# \ldots \#w_n\$w_n\# \ldots \#w_1 \mid w_1, \ldots, w_n \in \{0,1\}^* \,\}.$$

It is known that the language L_n is accepted by a k-head pushdown automaton if and only if $n \leq \binom{k}{2}$ [3,4]. Because of Corollary 14, this means that L_n is accepted by a centralized asynchronous PC system of pushdown automata of degree k that is working in returning mode if and only if $n \leq \binom{k}{2}$. This implies that the language

$$L_\infty = \{\, w_1\# \ldots \#w_n\$w_n\# \ldots \#w_1 \mid n \geq 1, w_1, \ldots, w_n \in \{0,1\}^* \,\}$$

is not accepted by any centralized asynchronous PC system of pushdown automata that is working in returning mode. The proof of Theorem 16 can now be completed by establishing the following claim.

Claim. L_∞ is accepted by a centralized asynchronous PC system of deterministic pushdown automata of degree 2 that is working in nonreturning mode. □

5 Concluding Remarks

We have introduced asynchronous variants of the PC systems of pushdown automata of [6] by using a special response symbol. In this way we obtained a characterization of the language classes defined by n-head (deterministic and nondeterministic) pushdown automata in terms of centralized asynchronous PC systems of pushdown automata that work in returning mode. In the nondeterministic case, these centralized systems accept all recursively enumerable languages, when they work in nonreturning mode. In addition, we could show that also in the deterministic case, centralized asynchronous PC systems of pushdown automata that work in nonreturning mode are more powerful than the systems of the same type that work in returning mode. However, we did not succeed in obtaining a characterization for the class of languages that are accepted by centralized asynchronous PC systems of deterministic pushdown automata that work in nonreturning mode. Given sufficiently many components, do these PC systems accept all recursively enumerable languages?

Concerning non-centralized APCPDA systems, it can be shown that each recursively enumerable language is accepted by an APCDPDA system of degree two that is working in nonreturning mode as well as by an APCDPDA system of degree three that is working in returning mode. These results have the following consequences.

Corollary 17. (a) $\mathcal{L}(\mathsf{APCDPDA}(n)) = \mathcal{L}(\mathsf{APCPDA}(n)) = \mathsf{RE}$ *for all* $n \geq 2$.
(b) $\mathcal{L}_r(\mathsf{APCDPDA}(n)) = \mathcal{L}_r(\mathsf{APCPDA}(n)) = \mathsf{RE}$ *for all* $n \geq 3$.

It remains to determine the expressive power of non-centralized asynchronous PC systems of (deterministic and nondeterministic) pushdown automata of degree two that work in returning mode.

In our definition a component of a PC system of pushdown automata sends its pushdown contents to each and every component that has the corresponding query symbol as the topmost symbol on its pushdown. It is, however, conceivable that a component may want to choose to which other component it is willing to send its pushdown contents. This could be achieved, for example, by using a set of response symbols $\{R_1, \ldots, R_n\}$ similar to the way in which communication is realized in PC systems of restarting automata as defined in [10]: a communication that sends the pushdown contents of component A_j to A_i can be executed only if the topmost symbol on the pushdown of A_i is the query symbol K_j, and the topmost symbol on the pushdown of A_j is the response symbol R_i. It appears, however, that all our results extend to this type of PC systems.

References

1. Balan, M.: Serializing the parallelism in parallel communicating pushdown automata systems. In: Dassow, J., Pighizzini, G., Truthe, B. (eds.) DCFS 2009. Electronic Proceedings in Theoretical Computer Science, vol. 3, pp. 59–68 (2009)
2. Buntrock, G., Otto, F.: Growing context-sensitive languages and Church-Rosser languages. Inform. and Comput. 141, 1–36 (1998)
3. Chrobak, M.: Hierarchies of one-way multihead automata languages. Theor. Comput. Sci. 48, 153–181 (1986)
4. Chrobak, M., Li, M.: $k+1$ heads are better than k for PDAs. J. Comp. Syst. Sci. 37, 144–155 (1988)
5. Csuhaj-Varjú, E., Dassow, J., Kelemen, J., Păun, G.: Grammar Systems - A Grammatical Approach to Distribution and Cooperation. Gordon and Breach, London (1994)
6. Csuhaj-Varjú, E., Martín-Vide, C., Mitrana, V., Vaszil, G.: Parallel communicating pushdown automata systems. Intern. J. Found. Comput. Sci. 11, 631–650 (2000)
7. Dassow, J., Păun, G., Rozenberg, G.: Grammar systems. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages. Linear Modelling: Background and Applications, vol. 2, pp. 101–154. Springer, Heidelberg (1997)
8. Otto, F.: Centralized PC Systems of Pushdown Automata versus Multi-head Pushdown Automata. In: Kutrib, M., Moreira, N., Reis, R. (eds.) DCFS 2012. LNCS, vol. 7386, pp. 244–251. Springer, Heidelberg (2012)
9. Petersen, H.: The power of centralized PC systems of pushdown automata. arXiv:1208.1283 (2012)
10. Vollweiler, M., Otto, F.: Systems of parallel communicating restarting automata. In: Freund, R., Holzer, M., Truthe, B., Ultes-Nitsche, U. (eds.) Fourth Workshop on Non-Classical Models for Automata and Applications (NCMA 2012). Proc., books@ocg.at, vol. 290, pp. 197–212. Oesterreichische Computer Gesellschaft, Wien (2012)

Model Checking Metric Temporal Logic over Automata with One Counter

Karin Quaas*

Institut für Informatik, Universität Leipzig,
04109 Leipzig, Germany
quaas@informatik.uni-leipzig.de

Abstract. We study the decidability status of the model checking problem for Metric Temporal Logic over models with one counter variable whose value can increase and decrease. This includes 1-counter machines with zero tests, 1-dimensional vector addition systems with states, and weighted automata with weights in the integers. We show that model checking of non-deterministic models is undecidable, even if we restrict the intervals used in the logic to be of the form $(-\infty, 0]$ and $[0, \infty)$. On the positive side, we show that model checking of deterministic models is decidable.

1 Introduction

During the last decades, *Metric Temporal Logic* (MTL, for short), introduced by Koymans [10], has become a prominent formalism for specifying the behaviour of real-time systems, like timed automata [1] and its weighted variants [5]. MTL extends Linear Temporal Logic (LTL, for short) by constraining the temporal operators by intervals of the non-negative real numbers. For instance, the formula $p \rightarrow \Diamond_{[2,3)} q$ expresses that whenever p holds, q should finally hold after 2 and before 3 time units, or, using an alternative interpretation of the variable, q should finally hold and the system's energy consumption should be in the interval $[2, 3)$. Recent research work shows that MTL-model checking over *finite* computations on models with *continuous* behaviour has non-primitive recursive complexity [14,5], and it is undecidable if one considers *infinite* computations over continuous models [13] or models with additional resources [5]. For conceptually simpler models with *discrete* behaviour, MTL-model checking is EXPSPACE-complete [2,11]. Recently [12], this positive decidability result could be generalized to MTL-model checking over *weighted automata* with weights coming from a monotone ordered weight structure that is bounded locally finite. Weight structures satisfying these properties comprise amongst others the set of non-negative rational numbers with the usual addition operation, and the set of rational numbers with the minimum operation; the set of integers with addition, however, does not satisfy the required properties. It was left as an open question

* The author was supported by *Deutsche Forschungsgemeinschaft (DFG)*, project QU 316/1-1.

in [12] whether MTL-model checking over weighted automata with weights in the integers is decidable or not.

In this paper, we answer this question negatively. We show by a reduction from the state-reachability problem for 2-counter machines that MTL-model checking over weighted automata with integer weights is undecidable. This is the case even if we restrict the logic to formulas where the intervals occurring as constraints at the temporal operators are of the form $(-\infty, 0]$ and $[0, \infty)$. The undecidability result also applies to the related models of 1-counter machines and 1-dimensional vector addition systems with states (VASS, for short). As a second result, we show that MTL-model checking becomes decidable if we restrict the models in consideration to be *deterministic*. For this, we generalize a result of Demri et al. [6] on the decidability of model checking the logic Freeze LTL over deterministic 1-counter machines.

Related work. It is folklore that LTL-model checking over deterministic 2-counter machines is undecidable: the proof can be done by an easy reduction from the halting problem for 2-counter machines. It is long known that LTL-model checking over n-dimensional VASS and pushdown systems is decidable [8,3]. For the recently established complexity results for 1-counter machines we refer the reader to [9].

The above mentioned results refer to classical LTL, which does not allow to express any properties about the value of the counter. In contrast to this, MTL permits to express differences between the values of a counter between two positions in a computation. This enables us to simulate zero tests and thus the behaviour of a 2-counter machine: hence MTL-model checking over non-deterministic 2-dimensional VASS and weighted automata over $\mathbb{Z}$ with 2 weight variables is undecidable.

Demri et al. [6] consider *Freeze LTL*, an extension of LTL that allows to test the values of a counter at two different positions in a computation for identity using a finite set of *registers*. Freeze LTL-model checking non-deterministic 1-counter machines is undecidable [6]; the same holds for 2-dimensional VASS, even for a restricted version of Freeze LTL [7]. To the best of our knowledge, there is no (un)decidability result for 1-dimensional VASS known. From a result by Bouyer et al. [4] it follows that MTL restricted to intervals of the form $[0, 0]$ and $\mathbb{Z}$ is strictly less expressive than Freeze LTL; hence the undecidability results for Freeze LTL do not automatically carry over to MTL.

On the positive side, model checking deterministic 1-counter machines against formulas of Freeze LTL is proved to be decidable [6]. However, the logic in [6] only allows to test whether the values of the counter at two different positions are identical, whereas MTL allows to test whether the distance between the values at two positions is in an arbitrary interval of $\mathbb{Z}$. Hence, the result in [6] cannot immediately be applied to MTL.

2 Counter Machines and Weighted Automata

A 1-*counter machine* is a tuple $M = (Q, q_0, \Delta)$, where Q is a finite set of states, $q_0 \in Q$ is an initial state, and $\Delta \subseteq Q \times Op \times Q$ is a finite set of transitions labeled with an operation from the set $Op := \{\texttt{++}, \texttt{--}, = 0?\}$. A *configuration* of M is a pair $(q, c) \in Q \times \mathbb{N}$. Between two configurations there is a *step*, denoted by $(q, c) \to (q', c')$, if, and only if, there is some transition (q, op, q'), and either

- op is $= 0?$ (zero test of the counter): $c = c' = 0$, and $d = d'$, or
- op is $\texttt{--}$ (decrementation of the counter): $c > 0$, $c' = c - 1$, and $d' = d$, or
- op is $\texttt{++}$ (incrementation of the counter): $c' = c + 1$, and $d' = d$.

A *finite computation* of M is a sequence $\gamma_0 \to \gamma_1 \to \dots \to \gamma_k$ of steps, where $\gamma_0 = (q_0, 0)$. An *infinite computation* of M is an infinite sequence $\gamma_0 \to \gamma_1 \to \dots$ of steps such that $\gamma_0 = (q_0, 0)$. A 1-counter machine is *deterministic* if for each reachable configuration γ there is *at most one* configuration γ' such that $\gamma \to \gamma'$ is a step. A 1-counter machine is a 1-*dimensional vector addition system with states* (VASS, for short), if its transitions do not use any zero tests.

A *weighted automaton* is a tuple $\mathcal{A} = (S, s_0, T, F)$, where S is a finite set of states, $s_0 \in S$ is the initial state, $T \subseteq S \times \{0, 1, -1\} \times S$ is a finite set of transitions, and $F \subseteq S$ is a set of accepting states. A *global state* of a weighted automaton is a pair $(q, c) \in S \times \mathbb{Z}$. Note that - in contrast to the values of the counters in counter machines - the value of the weight variable in a global state may become negative. A *finite run* of a weighted automaton is a finite sequence $(s_0, c_0)(s_1, c_1) \dots (s_k, c_k)$ of global states such that for each $i \in \{0, \dots, k-1\}$ there is a transition $(s_i, a_i, s_{i+1}) \in T$ such that $c_i + a_i = c_{i+1}$. *Infinite runs* are defined analogously. A weighted automaton $\mathcal{A}$ is *deterministic*, if for each $s \in S$ there is at most one transition $(s, a, s') \in T$ for some $a \in \{0, 1, -1\}$ and $s' \in S$.

3 Metric Temporal Logic

Given a finite set Q, the set of formulas of MTL is built up from Q by boolean connectives and constraining versions of the *next* and *until* operator as follows:

$$\varphi ::= \texttt{true} \mid q \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc_I \varphi \mid \varphi_1 \mathsf{U}_I \varphi_2$$

where $q \in Q$ and $I \subseteq \mathbb{Z}$ is an open, closed, or half-open interval with endpoints in $\mathbb{Z} \cup \{-\infty, \infty\}$. If $I = \mathbb{Z}$, then we omit the annotation I on $\bigcirc_I$ and U_I.

Note that in contrast to classical MTL [10], we allow for intervals containing negative integers.

Formulas in MTL are interpreted over computations of 1-counter machines and runs of weighted automata. Let $\gamma = (q_0, c_0) \to (q_1, c_1) \to \dots$ be an infinite computation of a 1-counter machine, and let $i \in \mathbb{N}$. We define the *satisfaction relation for MTL*, denoted by $\models$, inductively as follows:

$(\gamma, i) \models \texttt{true}$, $\quad (\gamma, i) \models q$ if $q = q_i$,

$(\gamma, i) \models \neg\varphi$ if $(\gamma, i) \not\models \varphi$,

$(\gamma, i) \models \varphi_1 \wedge \varphi_2$ if $(\gamma, i) \models \varphi_1$ and $(\gamma, i) \models \varphi_2$,

$(\gamma, i) \models \bigcirc_I \varphi$ if $(\gamma, i+1) \models \varphi$ and $c_{i+1} - c_i \in I$,

$(\gamma, i) \models \varphi_1 \mathsf{U}_I \varphi_2$ if $\exists j \geq i : (\gamma, j) \models \varphi_2, c_j - c_i \in I, \forall i \leq l < j : \ (\gamma, l) \models \varphi_1$.

In the same way, we define the satisfaction relation for MTL for finite computations of 1-counter machines, as well as for finite and infinite runs of weighted automata.

We use the following syntactical abbreviations: $\Diamond_I \varphi := \texttt{true}\mathsf{U}_I \varphi$, $\Box_I \varphi := \neg \Diamond_I \neg \varphi$.

We are interested in MTL-model checking counter machines and weighted automata; formally:

Infinitary Existential Model Checking Problem

INPUT: A 1-counter machine M (a weighted automaton $\mathcal{A}$, respectively), an MTL-formula φ.

QUESTION: Is there some infinite computation γ of M (infinite run γ of $\mathcal{A}$, respectively) such that $(\gamma, 0) \models \varphi$?

The *Finitary Existential Model Checking Problem* is defined in an analogous manner for finite computations of M (finite runs of $\mathcal{A}$, respectively).

4 Model Checking Non-deterministic Models

In this section, we present the undecidability result for MTL-model checking of *non-deterministic* weighted automata, VASS and 1-counter machines.

Theorem 1. *Finitary existential MTL-model checking of non-deterministic weighted automata is undecidable.*

Proof. The proof is by reduction from the undecidable state-reachability problem for deterministic 2-counter machines. A 2-counter machine is a tuple $M = (Q, q_0, \Delta)$, where $\Delta \subseteq Q \times Op' \times Q$ and $Op' = \{C_1, C_2\} \times \{\texttt{++}, \texttt{--}, =0?\}$, *i.e.*, the machine operates on two counters C_1 and C_2 instead of just one. A configuration of a 2-counter machine is thus a triple $(q, c, d) \in Q \times \mathbb{N}^2$. The definitions of *steps*, *computations* and *determinism* are analogous to those for 1-counter machines.

Let $M = (Q, q_0, \Delta)$ be a deterministic 2-counter machine, and let $q_F \in Q$. We define a weighted automaton $\mathcal{A}_M$ and an MTL-formula ψ_M such that M has a computation that reaches q_F if, and only if, there is a run γ of $\mathcal{A}_M$ such that $(\gamma, 0) \models \psi_M$. Next, we informally explain the idea of the reduction. A configuration (q, c, d) of M together with a transition $\delta = (q, op, q') \in \Delta$ is encoded by a run of $\mathcal{A}_M$ of the form

$$(q, c+d+1) \to (\delta^-, c+d) \to (\delta^-, c+d-1) \to \cdots \to (\delta^-, c) \to ((q, \delta), c),$$

i.e., the value of the weight variable in a state q corresponds to the sum of the values of the two counters of M plus 1 before the transition δ is executed, and the value of the weight variable in a state of the form (q, δ) represents the value of the first counter before the transition δ is executed.

Without loss of generality, we assume that q_0 has no ingoing transitions. We define the weighted automaton $\mathcal{A}_M = (S, s_0, T, F)$, where $S = Q \cup \{s_0\} \cup \{(q, \delta) \mid \delta \in \Delta, q \in Q\} \cup \{\delta^-, \delta^+ \mid \delta \in \Delta\}$, and $F = \{q_F\}$. For initializing $\mathcal{A}_M$, we define a transition $(s_0, 1, q_0)$. For each $\delta = (q, op, q') \in \Delta$, we define the transitions

- $(q, -1, \delta^-)$,
- $(\delta^-, 0, (q, \delta))$,
- $((q, \delta), 1, \delta^+)$,
- $(\delta^+, 1, \delta^+)$,
- $(\delta^+, 0, q')$.

If $q \neq q_0$, we additionally define the transition $(\delta^-, -1, \delta^-)$.

Next, we define the MTL-formula ψ_M to encode the correct semantics of the operations in M. The formula ψ_M is supposed to be evaluated at the initial configuration of M. We start with defining two helpful macros. Let $\delta = (q, op, q') \in \Delta$ and I be an interval. We define a macro $\psi_1(\delta, I)$ that expresses that the difference between the value of the weight variable in a state of the form (q, δ) and the very next state of the form (q', δ') for some δ' is in I. Since the value of the weight variable in a state (q, δ) represents the value of the first counter before the transition δ is executed, this formula expresses that the difference of the first counter after and before δ is executed is in I.

$$\Box\big[(q \wedge \bigcirc \delta^-) \rightarrow \bigvee_{\delta_2 = (q', op, q'') \in \Delta} \big((q \vee \delta^-) \mathsf{U} \big((q, \delta) \wedge (\bigcirc(\delta^+ \vee q' \vee \delta'^- \vee (q', \delta')) \mathsf{U}_I (q', \delta_2))\big)\big)\big].$$

Similarly, we define a macro $\psi_2(\delta, I)$ that expresses that the difference between the value of the weight variable in a state of the form q and the very next state of the form q' is in I. This refers to the difference of the sum of the values of the first and second counter plus 1 after and before δ is executed.

$$\Box[(q \wedge \bigcirc \delta^-) \rightarrow ((q \vee \delta^- \vee (q, \delta) \vee \delta^+) \mathsf{U}_I q')].$$

The definition of ψ_M is as follows:

- The value of the weight variable in a state of the form (q, δ) represents the value of the first counter, and thus should be non-negative. The following formula expresses that whenever the formula (q, δ) is true, the value of the weight variable is not in the interval $(-\infty, -1]$, *i.e.*, it is greater than or equal to 0:
$$\varphi_{\mathsf{Nonnegative}} := \bigwedge_{\delta = (q, -, -) \in \Delta} \Box_{(-\infty, -1]} \neg (q, \delta)$$
- A transition $\delta = (q, C_1 = 0?, q')$ can be executed only if the value of the first counter is zero. The value of the first counter is represented by the value of the weight variable in state (q, δ). The formula $\mathsf{ZeroTest1}(\delta)$ expresses that the value in (q, δ) is less than or equal to 0:
$$\mathsf{ZeroTest1}(\delta) := \Box_{[1, \infty)} \neg (q, \delta).$$
Together with $\varphi_{\mathsf{Nonnegative}}$ we obtain that the value in (q, δ) is equal to 0. We set
$$\varphi_{C_1 = 0?} := \bigwedge_{\delta = (q, C_1 = 0?, q') \in \Delta} \mathsf{ZeroTest1}(\delta) \wedge \mathsf{NoChange1}(\delta) \wedge \mathsf{NoChange2}(\delta),$$

where $\mathsf{NoChange1}(\delta) := \psi_1(\delta, [0,0])$ and $\mathsf{NoChange2}(\delta) := \psi_2(\delta, [0,0])$ (using the above defined macros) express that the value of the first and second counter are not changed by the execution of a zero test transition.

– A transition $\delta = (q, C_2 = 0?, q')$ can be executed only if the value of the second counter is zero. The value of the second counter is represented by the value of the weight variable in state q minus the value of the weight variable in the very next state of the form (q, δ) minus 1. The formula $\mathsf{ZeroTest2}(\delta)$ expresses that the value of the weight variable in state q and the value of the weight variable in state (q, δ) is exactly -1.

$$\mathsf{ZeroTest2}(\delta) := \Box[(q \wedge \bigcirc \delta^-) \to ((q \vee \delta^-)\mathsf{U}_{[-1,-1]}(q, \delta))]$$

This implies that the value of the second counter equals 0. We set

$$\varphi_{C_2=0?} := \bigwedge_{\delta=(q,C_2=0?,q')\in\Delta} \mathsf{ZeroTest2}(\delta) \wedge \mathsf{NoChange1}(\delta) \wedge \mathsf{NoChange2}(\delta).$$

– A transition $\delta = (q, C_1\texttt{--}, q')$ can be executed only if the value of the first counter is positive. The value of the first counter is represented by the value of the weight variable in state (q, δ). The following formula expresses that the value of the weight variable in state (q, δ) is not 0:

$$\mathsf{Positive1}(\delta) := \Box_{[0,0]}\neg(q, \delta)$$

This together with $\varphi_{\mathsf{Nonnegative}}$ implies that the value is positive. As an effect of the execution of δ, the value of the first counter has to be decreased by 1, whereas the value of the second counter must not change. Since the value of the first counter is represented in the weight values in both states of the form q and (q, δ), we define $\mathsf{Dec1}(\delta) := \psi_1(\delta, [-1,-1]) \wedge \psi_2(\delta, [-1,-1])$. Finally we set

$$\varphi_{C_1\texttt{--}} := \bigwedge_{\delta=(p,C_1\texttt{--},q)\in\Delta} \mathsf{Positive1}(\delta) \wedge \mathsf{Dec1}(\delta).$$

– A transition $\delta = (q, C_2\texttt{--}, q')$ can be executed only if the value of the second counter is positive. The value of the second counter is represented by the value of the weight variable in state q minus the value of the weight variable in the very next state of the form (q, δ) minus 1. The formula $\mathsf{Positive2}(\delta)$ expresses that the value of the weight variable in state q and the value of the weight variable in state (q, δ) is positive:

$$\mathsf{Positive2}(\delta) := \Box[(q \wedge \bigcirc \delta^-) \to ((q \vee \delta^-)\mathsf{U}_{(-\infty,-2]}(q, \delta))].$$

As an effect of the execution of δ, the value of the second counter has to be decreased by 1, whereas the value of the first counter must not change. We thus define $\mathsf{Dec2}(\delta) := \psi_2(\delta, [-1,-1])$, and set

$$\varphi_{C_2\texttt{--}} := \bigwedge_{(p,C_2\texttt{--},q)\in\Delta} \mathsf{Positive2}(\delta) \wedge \mathsf{NoChange1}(\delta) \wedge \mathsf{Dec2}(\delta).$$

– For incrementing transitions, the formulas have to express the increasing effects on the corresponding values. We define

$$\varphi_{c_1++} := \bigwedge_{(p,c_1++,q)\in\Delta} \mathsf{Inc1}(\delta),$$

where $\mathsf{Inc1}(\delta) := \psi_1(\delta_1, [1,1]) \wedge \psi_2(\delta_1, [1,1])$. Likewise, put

$$\varphi_{C_2\texttt{++}} := \bigwedge_{(p,C_2\texttt{++},q)\in\Delta} \mathsf{NoChange1}(\delta) \wedge \mathsf{Inc2}(\delta)$$

where $\mathsf{Inc2}(\delta) := \psi_2(\delta_1, [1,1])$.

– A run should satisfy ψ_M if, and only if, it finally reaches the state q_F. Hence set $\varphi_F = \Diamond q_F$.

Finally we set

$$\psi_M := \varphi_{\mathsf{Nonnegative}} \wedge \varphi_{C_1=0?} \wedge \varphi_{C_2=0?} \wedge \varphi_{C_1\texttt{--}} \wedge \varphi_{C_2\texttt{--}} \wedge \varphi_{C_1\texttt{++}} \wedge \varphi_{C_2\texttt{++}} \wedge \varphi_F.$$

□

We can easily adapt the proof of Theorem 1 and use 1-counter machines (VASS, respectively) to simulate the behaviour of a 2-counter machine:

Theorem 2. *Finitary existential MTL-model checking of non-deterministic 1-counter machines and VASS is undecidable.*

Remark 3. The definition of $\mathcal{A}_M$ and ψ_M in the proof of Theorem 1 can be changed to prove that MTL-model checking of weighted automata is undecidable even if the intervals I occurring in formulas $\bigcirc_I$ and U_I are restricted to be in $\{[0,\infty), (-\infty,0]\}$: For each transition $\delta = (q, op, q') \in \Delta$, we add six new states $q^1, q^2, q^3, (q^1,\delta), (q^2,\delta), (q^3,\delta)$, and define the transitions $(q, 1, q^1)$, $(q^1, -2, q^2)$, $(q^2, 1, q^3)$, $(q^3, -1, \delta^-)$, $(\delta^-, -1, \delta^-)$ (if $q \neq q_0$), $(\delta^-, 0, (q,\delta))$,$((q,\delta), 1, (q^1,\delta))$, $((q^1,\delta), -2, (q^2,\delta))$, $((q^2,\delta), 1, (q^3,\delta))$, $((q^3,\delta), 1, \delta^+)$, $(\delta^+, 1, \delta^+)$, $(\delta^+, 0, q')$. The formulas in ψ_M are slightly changed, *e.g.*, the formula $\varphi_{\mathsf{Nonnegative}}$ is redefined as $\bigwedge_{\delta\in\Delta} \Box_{(-\infty,0]} \neg(q^1,\delta)$.

5 Model Checking Deterministic Models

In this section, we prove that MTL-model checking is decidable if we restrict the considered models to be deterministic.

Theorem 4. *Infinitary existential MTL-model checking of deterministic* 1*-counter machines is decidable.*

The result immediately implies decidability of MTL-model checking of deterministic VASS and weighted automata.

The proof is based on a generalization of a result for model checking deterministic 1-counter machines against formulas of the logic Freeze LTL [6], (LTL$^{\downarrow}$, for short).

Given a finite set Q and a finite set R of *registers*, the set of formulas of LTL$^{\downarrow}$ is defined as follows:

$$\varphi ::= \texttt{true} \mid q \mid r \in I \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi \mid \varphi_1 \mathsf{U} \varphi_2 \mid \downarrow r.\varphi$$

where $q \in Q$, $r \in R$ and $I \subseteq Z$ is an interval. Note that this logic is a generalization of the logic defined in [6], which allows for intervals of the form $[0,0]$ only. We use LTL$_0^{\downarrow}$ to denote this logic. We use the syntactical abbreviation $\varphi_1 R \varphi_2 := \neg(\neg\varphi_1 \mathsf{U} \neg\varphi_2)$.

Formulas in LTL$^{\downarrow}$ are interpreted over computations of 1-counter machines. A *register valuation* ν is a function from R to $\mathbb{N}$. Let $\gamma = (q_0, c_0) \to (q_1, c_1) \to \dots$ be a computation of a 1-counter machine, let ν be a register valuation, and let $i \in \mathbb{N}$. The satisfaction relation for LTL$^{\downarrow}$, denoted by $\models_{\downarrow}$, is inductively defined as expected; we only give the definitions for the new formulas:

$$(\gamma, i, \nu) \models_{\downarrow} x \in I \text{ if } c_i - \nu(r) \in I,$$
$$(\gamma, i, \nu) \models_{\downarrow} \downarrow r.\varphi \text{ if } (\gamma, \nu[r \mapsto c_i], i) \models \varphi.$$

Here, $\nu[r \mapsto c_i]$ is the valuation that agrees with ν on all $r \in R \backslash \{r\}$, and maps r to c_i.

The following result is analogous to a result for the real-time logics MTL and TPTL [4].

Lemma 5. *For each MTL-formula φ there is a LTL$^{\downarrow}$-formula ψ such that for each computation $\gamma = (q_0, c_0) \to (q_1, c_1) \to \dots$ of a 1-counter machine, the empty valuation ν and $i \in \mathbb{N}$, we have $(\gamma, i) \models \varphi$ if, and only if, $(\gamma, \nu, i) \models_{\downarrow} \psi$.*

Theorem 4 follows from Lemma 5 and the next theorem, which is a generalization of Theorem 13 in [6].

Theorem 6. *Infinitary existential LTL$^{\downarrow}$-model checking of deterministic 1-counter machines is decidable.*

In the rest of this section we explain the main ideas of the proof of Theorem 6.

Let $M = (Q, q_0, \Delta)$ be a deterministic 1-counter machine. Note that M has at most one accepting computation, denoted by γ_M. In the following, we assume that γ_M is of the form $(q_0, c_0) \to (q_1, c_1) \to \dots$. The crucial point in the proof of Theorem 6 is the fact that γ_M can be decomposed into a regular form.

Lemma 7 ([6]). *There are K_1, K_2, K_3 such that $K_1 + K_2 \le |Q|^3$, $K_3 \le |Q|$, and for each $i \ge K_1$ we have $(q_{i+K_2}, c_{i+K_2}) = (q_i, c_i + K_3)$.*

Intuitively, γ_M consists of a finite prefix of length K_1, followed by a sequence of K_2 steps that are repeated infinitely often, and where the counter values increase in each iteration by K_3. For the rest of this section we let K_1, K_2 and K_3 be the smallest natural numbers that satisfy the conditions of Lemma 7. The interesting case is $K_3 > 0$, when the number of different values occurring in the computation is infinite. We assume in the following that $K_3 > 0$.

Let $\varphi \in$LTL$^{\downarrow}$. We use $\mathsf{sub}(\varphi)$ to denote the set of subformulas of φ, and we use $\mathsf{U}(\varphi)$ to denote the set of Until-subformulas of φ. We say that φ is in *negation* *normal form* if for every subformula $\neg\psi$ of φ either $\psi = \texttt{true}$, $\psi = q$ for some $q \in Q$ or $\psi = r \in I$ for some $r \in \mathsf{regs}(\varphi)$ and $I \subseteq \mathbb{Z}$, *i.e.*., negation is only applied at the atomic level. Note that there is a LTL$^{\downarrow}$-formula φ' in negation normal form (using formulas of the form $\varphi_1 R \varphi_2$) such that $(\gamma_M, i, \nu) \models_{\downarrow} \varphi$ iff $(\gamma_M, i, \nu) \models_{\downarrow} \varphi'$ for every $i \geq 0$ and register valuation ν.

We define $\mathsf{regs}(\varphi)$ to be the set of registers that occur in φ. For $r \in \mathsf{regs}(\varphi)$, define $\mathsf{ints}(r)$ to be the set of intervals I such that $r \in I$ is a subformula of φ. For technical reasons, we additionally require that $\mathsf{ints}(r)$ includes the interval $[0,0]$.

Let $i \geq 0$ and $m \in \mathbb{N}$. We define $\mathsf{offset}(i,m)$ to be the set $\{j \in \mathbb{N} \mid c_{i+j} = m\}$. A *context of* γ_M is a pair (i,ν), where $i \geq 0$ and ν is a register valuation ranging over $\{c_0, \ldots, c_i\}$. Next, we define an equivalence relation $\equiv$ over the set of contexts. For $(i,\nu),(i',\nu')$, we define $(i,\nu) \equiv (i',\nu')$ if, and only if, the following four conditions are satisfied.

- $q_i = q_{i'}$,
- $q_{i+\alpha} = q_{i+\beta}$ iff $q_{i'+\alpha} = q_{i'+\beta}$ for all $\alpha, \beta \geq 0$,
- $c_{i+\alpha} = c_{i+\beta}$ iff $c_{i'+\alpha} = c_{i'+\beta}$ for all $\alpha, \beta \geq 0$,
- $\mathsf{offset}(i,\nu(r)) = \mathsf{offset}(i',\nu'(r))$ for each $r \in \mathsf{regs}(\varphi)$.

In [6], the main ingredients of the decidability result are:

1. If $(i,\nu) \equiv (i',\nu')$ then $(\gamma_M, i, \nu) \models_{\downarrow} \psi$ if, and only if, $(\gamma_M, i', \nu') \models_{\downarrow} \psi$ for every $\psi \in$LTL$_0^{\downarrow}$.
2. The number of equivalence classes induced by $\equiv$ is finite.
3. Each equivalence class can be finitely represented by a *symbolic context*. The symbolic context of a context (i,ν) is a pair (i', srv), where $i' = i$ if $i < K_1$. Otherwise, i' is the unique element in $\{K_1, \ldots, K_1 + K_2 - 1\}$ such that i' is congruent modulo K_2 to i. Further, srv is a *symbolic register valuation* that maps each register r to $\mathsf{offset}(i,\nu(r))$. Note that $K_3 > 0$ implies that $\mathsf{offset}(i,\nu(r))$ is finite.

With these results, it is possible to reduce the infinitary LTL$_0^{\downarrow}$-model checking problem to a Büchi-acceptance problem for alternating Büchi automata. The idea is to construct an *alternating Büchi automaton* $\mathcal{A}_{M,\varphi}$ with states of the form $(\langle i,\nu\rangle, \psi)$, where $\langle i,\nu\rangle$ is a symbolic context representing the equivalence class of (i,ν), and $\psi \in \mathsf{sub}(\varphi)$. The automaton is constructed in such a way that there is a Büchi-accepting run of $\mathcal{A}_{M,\varphi}$ starting in the state $(\langle i,\nu\rangle,\psi)$ if, and only if, $(\gamma_M, i, \nu) \models_{\downarrow} \psi$.

In contrast to [6], we have to deal with arbitrary intervals I of $\mathbb{Z}$.

Let (i,ν) be a context, $m \in \mathbb{N}$ and let $I \subseteq Z$ be an interval. We define $\mathsf{intoffset}(i,m,I)$ to be the set $\{j \in \mathbb{N} \mid c_{i+j} - m \in I\}$. We note that $\mathsf{offset}(i,m) = \mathsf{intoffset}(i,m,[0,0])$. Now let $r \in \mathsf{regs}(\varphi)$ be a register such that $\nu(r)$ is defined. Obviously, for intervals $I \in \{[a,a],[a,b],(-\infty,a]\}$, $\mathsf{intoffset}(i,\nu(r),I)$ is the union of finitely many offsets; *e.g.*, $\mathsf{intoffset}(i,\nu(r),[a,b]) = \bigcup_{a \leq \alpha \leq b} \mathsf{offset}(i,\nu(r)+\alpha)$. For intervals of the form $[a,\infty)$ this is not the case. However, we prove that there is some $J \geq i$ such that $c_{i+k} - \nu(r) \in [a,\infty)$ for all $k > J$. Further, we prove that there is a bound B for the values that the counter can take in positions $i \leq k \leq J$. Thus, $\mathsf{intoffset}(i, \nu(r), [a, \infty)) = \bigcup_{a \leq \alpha \leq B} \mathsf{offset}(i, \nu(r) + \alpha) \cup \{k \mid k \geq J\}$, which can be finitely represented.

Lemma 8. *Let (i, ν) be a context, let $r \in \mathsf{regs}(\varphi)$ such that $\nu(r)$ is defined, let $I = [a, \infty)$, and let $k \geq i$.*

1. *If $i < K_1$, then $k \in \mathsf{intoffset}(i, \nu(r), I)$ if, and only if, $k \in \mathsf{offset}(i, \nu(r) + \alpha)$ for some $a \leq \alpha \leq B$, where $B = K_1 + K_1K_3 + aK_3 + K_2/2 + c_i$, or $k > K_1 + K_1K_2 + aK_2$.*
2. *If $i \geq K_1$, then $k \in \mathsf{intoffset}(i, \nu(r), I)$, if, and only if, $k \in \mathsf{offset}(i, \nu(r) + \alpha)$ for some $a \leq \alpha \leq B$, where $B = K_2K_3 + aK_3 + K_2/2 + c_i$, or $k > K_2^2 + aK_2$.*

This can be used to prove the next lemma.

Lemma 9. *Let $(i, \nu), (i', \nu')$ be two contexts. If $(i, \nu) \equiv (i', \nu')$, then we have $\mathsf{intoffset}(i, \nu(r), I) = \mathsf{intoffset}(i', \nu'(r), I)$ for each interval $I \subseteq Z$ and $r \in \mathsf{regs}(\varphi)$ such that $\nu(r), \nu'(r)$ are defined.*

We redefine the notion of *symbolic register valuation.* A symbolic register valuation is a mapping that maps each register $r \in \mathsf{regs}(\varphi)$ to a function from the set $\mathsf{ints}(r)$ to a set of positions in γ_M. Given (i, ν), we define $\mathsf{srv}(r)(I) = \mathsf{intoffset}(i, \nu(r), I)$ for every $r \in \mathsf{regs}(\varphi)$ and $I \in \mathsf{ints}(r)$. A context (i, ν) is represented by a *symbolic context*, which is a pair (i', srv) defined as above, but using our new definition of symbolic register valuations. We use $\langle i, \nu \rangle$ to denote the symbolic context of (i, ν).

Lemma 10. *Let $(i, \nu), (i', \nu')$ be two contexts. Then we have $(i, \nu) \equiv (i', \nu')$ if, and only if, $\langle i, \nu \rangle = \langle i', \nu' \rangle$.*

Altogether, we can show the following:

1. If $(i, \nu) \equiv (i', \nu')$ then $(\gamma_M, i, \nu) \models_{\downarrow} \psi$ if, and only if, $(\gamma_M, i', \nu') \models_{\downarrow} \psi$ for every $\psi \in \mathrm{LTL}^{\downarrow}$. This can be proved using Lemma 9 and Lemma 9(I) in [6].
2. The number of equivalence classes induced by $\equiv$ is finite.
3. Each equivalence class can be finitely represented by symbolic contexts. This follows from Lemmas 8 and 10.

Without loss of generality, we may assume that φ is in negation normal form. We define the alternating Büchi automaton $\mathcal{A}_{M,\varphi} = (S, s_0, \delta, F)$ as follows.

- $S = \{((i, \mathsf{srv}), \psi) \mid (i, \mathsf{srv})$ is a symbolic context, $\psi \in \mathsf{sub}(\varphi)\}$,
- $s_0 = \{((0, \mathsf{srv}_0), \varphi)\}$, where srv_0 is a symbolic register valuation representing the empty register valuation,
- $F = \{((-, -), \psi) \mid \psi \notin \mathsf{U}(\varphi)\}$,

– and δ is defined as follows:

$$\delta((i,\mathsf{srv}),q) = \begin{cases} \top & \text{if } q_i = q \\ \bot & \text{otherwise} \end{cases} \qquad \delta((i,\mathsf{srv}),\neg q) = \begin{cases} \bot & \text{if } q_i = q \\ \top & \text{otherwise} \end{cases}$$

$$\delta((i,\mathsf{srv}),r \in I) = \begin{cases} \top & \text{if } 0 \in \mathsf{srv}(r)(I) \\ \bot & \text{otherwise} \end{cases}$$

$$\delta((i,\mathsf{srv}),\neg r \in I) = \begin{cases} \bot & \text{if } 0 \in \mathsf{srv}(r)(I) \\ \top & \text{otherwise} \end{cases}$$

$$\delta((i,\mathsf{srv}),\psi \wedge \psi') = \delta((i,\mathsf{srv}),\psi) \wedge \delta((i,\mathsf{srv}),\psi')$$
$$\delta((i,\mathsf{srv}),\psi \vee \psi') = \delta((i,\mathsf{srv}),\psi) \vee \delta((i,\mathsf{srv}),\psi')$$
$$\delta((i,\mathsf{srv}),\bigcirc\psi) = (\mathsf{next}(i,\mathsf{srv}),\psi)$$
$$\delta((i,\mathsf{srv}),\downarrow r.\psi) = \delta((i, \bigcup_{I\in\mathsf{ints}(r)} \mathsf{srv}(r)(I) \mapsto \mathsf{offset}(i,c_i,I)),\psi)$$
$$\delta((i,\mathsf{srv}),\psi\mathsf{U}\psi') = \delta((i,\mathsf{srv}),\psi') \vee [\delta((i,\mathsf{srv}),\psi) \wedge \mathsf{next}((i,\mathsf{srv}),\psi\mathsf{U}\psi')]$$
$$\delta((i,\mathsf{srv}),\psi R\psi') = \delta((i,\mathsf{srv}),\psi') \wedge [\delta((i,\mathsf{srv}),\psi) \vee \mathsf{next}((i,\mathsf{srv}),\psi R\psi')]$$

Here, like in [6], the function next takes as input a symbolic context (i,srv) and returns the symbolic context (i',srv') satisfying the following conditions.

– If $i < K_1 + K_2 - 1$, then $i' = i - 1$; otherwise $i' = K_1$.
– For every $r \in \{1,\ldots,N\}$ and $I \in \mathsf{ints}(r)$ we put $\mathsf{srv}'(r)(I) = \{\alpha - 1 \mid \alpha \in \mathsf{srv}(r)(I), \alpha > 0\}$.

We can prove the following lemma similarly to Lemma 12 in [6], finishing the proof of Theorem 6.

Lemma 11. *Let (i,ν) be a context. For every $\psi \in \mathsf{sub}(\varphi)$, $(\gamma_M,i,\nu) \models_\downarrow \psi$ if, and only if, there is some Büchi-accepting run from $(\langle i,\nu\rangle,\psi)$ in $A_{M,\varphi}$.*

6 Conclusion and Open Questions

In the first part of this paper, we have proved undecidability of MTL-model checking of non-deterministic models. This together with Lemma 5 also implies the undecidability of model checking Freeze LTL over non-deterministic 1-dimensional VASS, which, to the best of our knowledge, was unknown so far. It is interesting to further investigate the differences between the logics MTL and Freeze LTL; *e.g.*, is MTL-model checking over non-deterministic 1-counter machines still undecidable if we restrict the intervals to be of the form $[0,0]$, as is done in original Freeze LTL [6]? In general, how can we restrict MTL in such a way that model checking becomes decidable for the computationally less expressive models of weighted automata and VASS? Concerning the decidability result for model checking of deterministic models in the second part of this paper, one may further investigate whether the complexity results for $\mathrm{LTL}_0^\downarrow$ in [6] carry over to $\mathrm{LTL}^\downarrow$.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
2. Alur, R., Henzinger, T.A.: A really temporal logic. J. ACM 41(1), 181–204 (1994)
3. Bouajjani, A., Esparza, J., Maler, O.: Reachability Analysis of Pushdown Automata: Application to Model-Checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
4. Bouyer, P., Chevalier, F., Markey, N.: On the expressiveness of TPTL and MTL. Inf. Comput. 208(2), 97–116 (2010)
5. Bouyer, P., Larsen, K.G., Markey, N.: Model checking one-clock priced timed automata. Logical Methods in Computer Science 4(2) (2008)
6. Demri, S., Lazić, R., Sangnier, A.: Model Checking Freeze LTL over One-Counter Automata. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 490–504. Springer, Heidelberg (2008)
7. Demri, S., Sangnier, A.: When Model-Checking Freeze LTL over Counter Machines Becomes Decidable. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 176–190. Springer, Heidelberg (2010)
8. Esparza, J.: Decidability and Complexity of Petri Net Problems - An Introduction. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 374–428. Springer, Heidelberg (1998)
9. Göller, S., Haase, C., Ouaknine, J., Worrell, J.: Model Checking Succinct and Parametric One-Counter Automata. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 575–586. Springer, Heidelberg (2010)
10. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2(4), 255–299 (1990)
11. Laroussinie, F., Markey, N., Schnoebelen, P.: On Model Checking Durational Kripke Structures. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 264–279. Springer, Heidelberg (2002)
12. Meinecke, I., Quaas, K.: Parameterized model checking of weighted networks, submitted to: TCS special issue WATA (2012)
13. Ouaknine, J., Worrell, J.B.: On Metric Temporal Logic and Faulty Turing Machines. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 217–230. Springer, Heidelberg (2006)
14. Ouaknine, J., Worrell, J.: On the decidability and complexity of metric temporal logic over finite words. Logical Methods in Computer Science 3(1) (2007)

Coinductive Proof Techniques for Language Equivalence

Jurriaan Rot[1,2,⋆], Marcello Bonsangue[1,2], and Jan Rutten[2,3]

[1] LIACS – Leiden University, Niels Bohrweg 1, Leiden, The Netherlands
[2] Centrum Wiskunde en Informatica, Science Park 123, Amsterdam, The Netherlands
[3] ICIS – Radboud University Nijmegen, Heyendaalseweg 135, Nijmegen The Netherlands
`{jrot,marcello}@liacs.nl, janr@cwi.nl`

Abstract. Language equivalence can be checked coinductively by establishing a bisimulation on suitable deterministic automata. We improve and extend this technique with *bisimulation-up-to*, which is an enhancement of the bisimulation proof method. First, we focus on the regular operations of union, concatenation and Kleene star, and illustrate our method with new proofs of classical results such as Arden's rule. Then we extend our enhanced proof method to incorporate language complement and intersection. Finally we define a general format of behavioural differential equations, in which one can define operations on languages for which bisimulation-up-to is a sound proof technique.

1 Introduction

The set of all languages over a given alphabet can be turned into an (infinite) deterministic automaton. By the *coinduction* principle, any two languages that are *bisimilar* as states in this automaton are in fact equal. The typical way to show that two languages x and y are bisimilar is by exhibiting a *bisimulation*, which in this setting is a relation on languages containing the pair (x, y) and satisfying certain properties. Indeed, this is the basis of a practical coinductive proof method for language equality [21], which has, for example, been applied in effective procedures for checking equivalence of regular languages [12,21,13,6].

In this paper we present *bisimulation up to congruence*, in the context of languages and automata. This is an enhancement of bisimulation originally stemming from process theory [24,18]. In order to prove bisimilarity of two languages, instead of showing that they are related by a bisimulation, one can show that they are related by a *bisimulation-up-to*, which in many cases yields smaller, easier and more elegant proofs. As such, we introduce a proof method which improves on the more classical coinductive approach based on bisimulations.

At first, we will focus on languages presented by the regular operations of union, concatenation and Kleene star. We will exemplify our coinductive proof

⋆ The research of this author has been funded by the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920.

method based on bisimulation-up-to by novel proofs of several classical results such as Arden's rule and the soundness of the axioms of Kleene algebra. Then we proceed to incorporate language intersection and complement and show the usefulness and versatility of the techniques by giving a full coinductive proof that two context-free languages defined in terms of language equations, that of palindromes and that of non-palindromes, indeed form each others complement. Finally, in order to deal with other language operations, such as shuffle or symmetric difference, we introduce a general format of *behavioural differential equations*, for operations allowing proofs based on bisimulation-up-to.

The soundness of bisimulation-up-to, stating that whenever two languages are related by a bisimulation-up-to they are equal, follows from abstract coalgebraic theory [20,19]. The main contribution of this paper is the presentation of this proof technique in the context of languages and automata, without explicitly using any coalgebra or category theory. As witnessed by the many examples in this paper, this yields a useful, elegant and efficient method for proving equality of languages, enhancing the existing successful coinductive methods based on establishing bisimulations. Moreover, from the perspective of coalgebraic theory this can be regarded as an extensive concrete exercise in bisimulation-up-to. On the technical side, the general format of operations for which bisimulation-up-to is sound, can be considered a novel contribution.

The outline of this paper is as follows. In Section 2 we recall the notions of bisimulation and coinduction in the context of languages and automata. Then in Section 3 we present bisimulation-up-to for the regular operations. In Section 4 we extend our techniques to deal with complementation and intersection, and in Section 5 we present a format for which bisimulation-up-to is guaranteed to be sound. In Section 6 we place our work in the context of coalgebraic theory and discuss related work, and finally in Section 7 we conclude.

2 Languages, Automata, Bisimulations and Coinduction

Throughout this paper we assume a fixed alphabet A, which is simply a (possibly infinite) set. We denote by A^* the set of *words*, i.e., finite concatenations of elements of A; we denote the empty word by ε, and the concatenation of two words w and v by wv. The set of *languages* over A is given by $\mathcal{P}(A^*)$, and ranged over by x, y, z. We denote the empty language by 0 and the language $\{\varepsilon\}$ by 1. Moreover when no confusion is likely to arise, we write a to denote the language $\{a\}$, for alphabet letters $a \in A$.

A *(deterministic) automaton* over A is a triple (S, o, δ) where S is a set of states, $o\colon S \to \{0,1\}$ is an output function, and $\delta\colon S \times A \to S$ is a transition function. Notice that S is not necessarily finite, and there is no initial state. We say a state $s \in S$ is *final* or *accepting* if $o(s) = 1$. For each automaton (S, o, δ) there is a function $l\colon S \to \mathcal{P}(A^*)$ which assigns to each state $s \in S$ a language, inductively defined as follows: $\varepsilon \in l(s)$ iff $o(s) = 1$ and $aw \in l(s)$ iff $w \in l(\delta(s, a))$.

The classical definition of *bisimulation* [14,17] applies to labelled transition systems, which, in contrast to deterministic automata, do not feature output and may have a non-deterministic branching behaviour. We will base ourselves on a different notion of bisimulation specific to deterministic automata, which is an instantiation of general coalgebraic theory (see Section 6). A *bisimulation* is a relation $R \subseteq S \times S$ such that for any $(s,t) \in R$: $o(s) = o(t)$ and for all $a \in A$: $(\delta(s,a), \delta(t,a)) \in R$. Given a deterministic automaton, the union of all bisimulations is again a bisimulation, is denoted by $\sim$ and is called *bisimilarity*; if $s \sim t$ for two states s,t then we say these states are *bisimilar*. Notice that in order to show that two states s and t are bisimilar, it suffices to construct a bisimulation R such that $(s,t) \in R$. As it turns out, this gives a proof principle for showing language equivalence of states:

Theorem 1 (Coinduction). *For any two states s,t of a deterministic automaton: if $s \sim t$ then $l(s) = l(t)$.*

This coinduction principle is easily proved by induction. The converse holds as well, i.e., if $l(s) = l(t)$ then s is related to t by some bisimulation R. Such a relation R may well be infinite, but this is not necessarily a problem; in practice one can often give a finite description of such an infinite relation.

In order to proceed we recall the notion of *language derivatives*: the a-derivative of a language x is defined as $x_a = \{w \mid aw \in x\}$. The set $\mathcal{P}(A^*)$ of all languages can be turned into a deterministic automaton by defining the output function and the transition function as follows: $o(x) = 1$ if $\varepsilon \in x$ and $o(x) = 0$ otherwise, and $\delta(x,a) = x_a$ for all $a \in A$. One can check that for any language x, the language accepted by the corresponding state in the automaton is precisely x itself. A relation R on languages is a bisimulation on this automaton if for any $(x,y) \in R$: $o(x) = o(y)$ and for any $a \in A$: $(x_a, y_a) \in R$. By the coinduction principle (Theorem 1), we now have the following method for checking equality of languages x and y: if we can establish a bisimulation containing the pair (x,y), then $x \sim y$, so $x = y$.

We will be interested in the regular operations on languages, defined in a standard way: union $x + y = \{w \mid w \in x \text{ or } w \in y\}$, concatenation $x \cdot y = \{w \mid w = uv \text{ for some } u \in x \text{ and } v \in y\}$ and Kleene star $x^* = \sum_{i \geq 0} x^i$, where $x^0 = 1$ and $x^{i+1} = x \cdot x^i$. We often write xy for $x \cdot y$.

In order to prove equivalence of languages defined using the above operations we may use bisimulations, but for this we need a characterization of the output (acceptance of the empty word) and the derivatives of languages. Such a characterization was given for regular expressions by Brzozowski [3]; we formulate this in terms of languages (e.g., [5, page 41]):

Lemma 2. *For any two languages x,y and for any $a,b \in A$:*

$$\begin{array}{ll}
0_a = 0 & o(0) = 0 \\
1_a = 0 & o(1) = 1 \\
b_a = \begin{cases} 1 & \text{if } b = a \\ 0 & \text{otherwise} \end{cases} & o(b) = 0 \\
(x+y)_a = x_a + y_a & o(x+y) = o(x) \vee o(y) \\
(x \cdot y)_a = x_a \cdot y + o(x) \cdot y_a & o(x \cdot y) = o(x) \wedge o(y) \\
(x^*)_a = x_a \cdot x^* & o(x^*) = 1
\end{array}$$

Example 3. Let us prove that $(a+b)^* = (a^*b^*)^*$ for some alphabet letters a, b (for simplicity we assume that the alphabet does not contain any other letters). To this end, we start with the relation $R = \{((a+b)^*, (a^*b^*)^*)\}$ and try to show it is a bisimulation. So we must show that the outputs of $(a+b)^*$ and $(a^*b^*)^*$ coincide, and that their a-derivatives and their b-derivatives are related by R. Using Lemma 2, we see that $o((a+b)^*) = 1 = o((a^*b^*)^*)$. Moreover, again using Lemma 2, we have $((a+b)^*)_a = (a+b)_a(a+b)^* = (1+0)(a+b)^* = (a+b)^*$ and $((a^*b^*)^*)_a = (a^*b^*)_a(a^*b^*)^* = ((a^*)_a b^* + o(a^*)(b^*)_a)(a^*b^*)^* = (a^*b^* + 0)(a^*b^*)^* = (a^*b^*)^*$, so the a-derivatives are again related (notice that apart from Lemma 2, we have used some basic facts about the regular operations). The b-derivative of $(a+b)^*$ is $(a+b)^*$ itself; however, the b-derivative of $(a^*b^*)^*$ is $b^*(a^*b^*)^*$. This means that R is not a bisimulation. However, we can consider instead the relation $R' = R \cup \{((a+b)^*, b^*(a^*b^*)^*)\}$. As it turns out, the pair $((a+b)^*, b^*(a^*b^*)^*)$ satisfies the necessary conditions as well, turning R' into a bisimulation. We leave the details as an exercise for the reader, and conclude $(a+b)^* = (a^*b^*)^*$ by coinduction.

Bisimulation proofs in general will follow the above pattern of using Lemma 2 to compute outputs and to expand the derivatives, and then using some reasoning to show that the outputs are equal and the derivatives related. In the sequel we will sometimes use Lemma 2 without further reference to it. We note that the above coinductive proof method applies to general languages, not only to regular ones. However if one restricts to regular languages, then this technique give rise to an effective algorithm for checking equivalence.

The axioms of *Kleene algebra (KA)* [11] constitute a complete axiomatisation of language equivalence of regular expressions. We recall them here for the following two reasons. First, they provide a number of interesting examples for our methods. Second, and more importantly, these axioms are often quite useful for relating derivatives. We state the axioms in terms of languages and our concrete operations:

$$
\begin{array}{lr@{\qquad}lr}
x + (y+z) = (x+y) + z & (1) & (y+z)x = yx + zx & (8) \\
x + y = y + x & (2) & x(y+z) = xy + xz & (9) \\
x + x = x & (3) & x^*x + 1 = x^* & (10) \\
x + 0 = x & (4) & xx^* + 1 = x^* & (11) \\
x(yz) = (xy)z & (5) & xy \subseteq x \to xy^* \subseteq x & (12) \\
x \cdot 1 = 1 \cdot x = x & (6) & yx \subseteq x \to y^*x \subseteq x & (13) \\
x \cdot 0 = 0 \cdot x = 0 & (7) & &
\end{array}
$$

Notice that $x \subseteq y$ iff $x + y = y$. All of (1) through (9) follow easily from the definition of the operations. The remaining axioms will be treated below.

3 Bisimulation-up-to

In this section we will introduce an enhancement of the bisimulation proof method. We first illustrate the need for such an enhancement with an example. Consider the Kleene algebra axiom (11) (see Section 2). In order to prove this identity coinductively, consider the relation $R = \{(xx^*+1, x^*) \mid x \in \mathcal{P}(A^*)\}$; let us see if this is a bisimulation. Using Lemma 2, it is easy to show that for any language x, the outputs of $xx^* + 1$ and x^* are equal. For any $a \in A$:

$$(xx^* + 1)_a = x_a x^* + o(x)x_a x^* + 0 = x_a x^* = (x^*)_a$$

where the leftmost and rightmost equality are by Lemma 2, and in the second step we use some of the KA identities which we know to hold. Now we have shown that the derivatives are *equal*; this does not allow us to conclude that R is a bisimulation, since for that, the derivatives need to be *related by* R. Instead we can consider the relation $R' = R \cup \{(y, y) \mid y \in \mathcal{P}(A^*)\}$. Then the derivatives of $xx^* + 1$ and x^* are related by R'; moreover, the diagonal is easily seen to satisfy the properties of a bisimulation as well. This solves the problem, but is obviously somewhat inconvenient.

Now consider the relation $R = \{(x^*x + 1, x^*) \mid x \in \mathcal{P}(A^*)\}$ which we may try to use to prove (10) by coinduction. The derivatives are (using Lemma 2):

$$(x^*x + 1)_a = x_a x^* x + x_a + 0 = x_a(x^*x + 1) \qquad \text{and} \qquad (x^*)_a = x_a x^*$$

Clearly $x_a x^*$ can be obtained from $x_a(x^*x + 1)$ by substituting $x^*x + 1$ for x^*, and indeed the latter two languages are related by R. However, unfortunately these derivatives are not related *directly* by R, and so R is not a bisimulation. Extending R to a bisimulation is indeed a non-trivial task.

When proving identities over languages coinductively, situations such as in the above examples occur very often. To deal with such cases in a better way, we will introduce *bisimulation-up-to.* We need the notion of congruence closure.

Definition 4. *For a relation $R \subseteq \mathcal{P}(A^*) \times \mathcal{P}(A^*)$, define the* congruence closure *of R as the least relation $\equiv$ satisfying the following rules:*

$$\frac{xRy}{x \equiv y} \qquad \frac{}{x \equiv x} \qquad \frac{x \equiv y}{y \equiv x} \qquad \frac{x \equiv y \qquad y \equiv z}{x \equiv z}$$

$$\frac{x_1 \equiv y_1 \qquad x_2 \equiv y_2}{x_1 + x_2 \equiv y_1 + y_2} \qquad \frac{x_1 \equiv y_1 \qquad x_2 \equiv y_2}{x_1 \cdot x_2 \equiv y_1 \cdot y_2} \qquad \frac{x \equiv y}{x^* \equiv y^*}$$

In the sequel we denote the congruence closure of a given relation R by $\equiv_R$.

The upper left rule ensures $R \subseteq \equiv_R$. The three rules on the right in the first row ensure that $\equiv_R$ is an equivalence relation. Notice that the reflexivity rule has as a consequence that languages which are equal, are also related by $\equiv_R$. The transitivity rule allows to relate languages in multiple "proof steps". Finally the three rules on the second row ensure that $\equiv_R$ is a congruence, which in particular means that $\equiv_R$ relates languages which are obtained by (syntactic) substitution of languages related by R. For example, if $x^*x + 1 \; R \; x^*$, then we can derive from the above rules that $x_a(x^*x + 1) \equiv_R x_a x^*$.

Definition 5. *Let $R \subseteq \mathcal{P}(A^*) \times \mathcal{P}(A^*)$ be a relation on languages. We say R is a* bisimulation up to congruence*, or simply a* bisimulation-up-to*, if for any pair $(x, y) \in R$: $o(x) = o(y)$ and for all $a \in A$: $x_a \equiv_R y_a$.*

The idea of bisimulation-up-to is that now the derivatives are easier to relate, since they can be related by the *congruence* $\equiv_R$ instead of only the relation R itself. Indeed, to prove that R is a bisimulation-up-to, the derivatives can be related by equational reasoning. Of course such a bisimulation-up-to R is in general not a bisimulation. As it turns out, showing that R is a bisimulation-up-to is enough to ensure that $\equiv_R$ is itself a bisimulation (this result is based on general coalgebraic theory which we will shortly discuss in Section 6). This means that in that case for any $(x, y) \in \equiv_R$ we have $x = y$ by coinduction. Since $R \subseteq \equiv_R$ this holds in particular for all pairs related by the bisimulation-up-to R. So we have the following proof principle:

Theorem 6 (Coinduction-up-to). *If R is a bisimulation-up-to then for any $(x, y) \in R$: $x = y$.*

Any bisimulation is also a bisimulation-up-to, so this generalizes Theorem 1 in the case of languages. Consequently, the converse of the above principle holds as well. We proceed with a number of proofs based on bisimulation-up-to.

Example 7. Recall the relation $R = \{(x^*x + 1, x^*) \mid x \in \mathcal{P}(A^*)\}$ from the beginning of this section. As we have seen, the a-derivatives are $x_a(x^*x + 1)$ and $x_a x^*$, which are not related by R; however they *are* related by $\equiv_R$. So R is a bisimulation-up-to, and consequently $x^*x + 1 = x^*$. Moreover, the relation $\{(xx^* + 1, x^*) \mid x \in \mathcal{P}(A^*)\}$ from the beginning of this section is a bisimulation-up-to as well; there, the derivatives are equal and thus related by $\equiv_R$.

We have covered coinductively the soundness of most of the axioms of Kleene algebra; the only remaining ones are right and left star induction, i.e., (12) and (13). Instead of proving these we will consider *Arden's rule* below; the coinductive proof of (12) and (13) is very similar (notice that we can treat an inclusion $x \subseteq y$ simply by showing $x + y = y$). Finally, notice that instead of proving the KA axioms one by one, we could consider the equivalence relation R induced by all the KA axioms together, and show that this is a bisimulation-up-to.

Example 8. Arden's rule states that if $x = yx + z$ for some languages x, y and z, and y does not contain the empty word, then $x = y^*z$. In order to prove its validity coinductively, let x, y, z be languages such that $\varepsilon \notin y$ and $x = yx + z$, and let $R = \{(x, y^*z)\}$. Then $o(y) = 0$, so $o(x) = o(yx + z) = (o(y) \wedge o(x)) \vee o(z) = (0 \wedge o(x)) \vee o(z) = o(z) = 1 \wedge o(z) = o(y^*) \wedge o(z) = o(y^*z)$ and for any $a \in A$: $x_a = (yx + z)_a = y_a x + o(y) \cdot x_a + z_a = y_a x + z_a \equiv_R y_a y^* z + z_a = (y^*z)_a$. So R is a bisimulation-up-to, proving Arden's rule.

While Arden's rule is not extremely difficult to prove without coinduction either, the textbook proofs are significantly longer and arguably more involved than the above proof, which is not much more than taking derivatives combined with a tiny bit of algebraic reasoning. Nevertheless this coinductive proof is completely precise. Giving a formal proof without our methods is not trivial at all; see, e.g., [7] for the discussion of a proof within the theorem prover Isabelle.

In fact, [21] already contains a coinductive proof of Arden's rule; however, this is based on a bisimulation (in contrast to our proof which is based on a bisimulation-up-to). Indeed, in [21] the infinite relation $\{(ux+v, uy^*z+v) \mid u, v \in \mathcal{P}(A^*)\}$ is used, requiring more work in checking the bisimulation conditions. In that case one essentially closes the relation under (certain) contexts manually; the coinduction-up-to principle does this in a general and systematic fashion.

Example 9. Let us prove that for any language x, we have $xx = 1 \to x = 1$. Assume $xx = 1$ and let $R = \{(x, 1)\}$. Since $o(xx) = o(1) = 1$ also $o(x) = 1 = o(1)$. We show that the derivatives of x and 1 are equal, turning R into a bisimulation-up-to. For any $a \in A$: $x_a x + x_a = x_a x + o(x)x_a = (xx)_a = 1_a = 0$. One easily proves that this implies $x_a = 0$ (for example by showing that $\{(y, 0) \mid x + y = 0\}$ is a bisimulation). Thus $x_a = 0 = 1_a$, so $x_a \equiv_R 1_a$.

Example 10. We prove the soundness of the axiom $xx = x \to x^* = 1 + x$, by establishing a bisimulation-up-to. This axiom was used by Boffa in his complete axiomatisation of equivalence of regular expressions. Let x be a language for which $xx = x$ and consider the relation $R = \{(x^*, 1 + x)\}$. Indeed, $o(x^*) = 1 = o(1 + x)$, and for any $a \in A$: $(x^*)_a = x_a x^* \equiv_R x_a(1 + x) = x_a + x_a x = x_a + o(x)x_a + x_a x = x_a + (xx)_a = x_a + x_a = x_a$.

In fact the above axiom can also easily be proved by induction. In practice, one wants to combine inductive and coinductive methods.

4 Language Equations and Boolean Operators

Context-free languages can be expressed in terms of certain types of language equations [8]. For example, the language $\{a^n b^n \mid n \in \mathbb{N}\}$ is the unique language x such that $x = axb + 1$. Our coinductive techniques can directly be applied to languages defined in such a way, and so we are able to reason about (equivalence of) context-free languages in a novel manner.

Example 11. Let x, y, z be languages such that $x = ayzb + 1$, $y = azxb + 1$ and $z = axyb + 1$. Without thinking of what possible concrete descriptions of x, y and z can be, let us show, by coinduction, that $x = y = z$. We use the relation $R = \{(x, y), (y, z)\}$. Obviously $o(x) = o(y)$ and $o(y) = o(z)$. Moreover for any alphabet letter b other than a, we have $x_b = 0 = y_b$ and $y_b = 0 = z_b$. For the a-derivatives we have $x_a = yzb \equiv_R zyb \equiv_R zxb = y_a$ and similarly for (y, z); so R is a bisimulation-up-to, proving that $x = y = z$.

So far we have considered the regular operations of union, concatenation and Kleene star. We proceed to incorporate complement and intersection, defined as $\overline{x} = \{w \mid w \notin x\}$ and $x \wedge y = \{w \mid w \in x \text{ and } w \in y\}$ respectively. Language equations including these additional operators can be used to give semantics to *conjunctive*- and *Boolean grammars* [16]. Complement and intersection have a known characterization in terms of outputs and derivatives as well [3]:

Lemma 12. *For any two languages x, y and for any $a \in A$:*

$$o(\overline{x}) = \neg o(x) \qquad \overline{x}_a = \overline{x_a}$$
$$o(x \wedge y) = o(x) \wedge o(y) \qquad (x \wedge y)_a = x_a \wedge y_a$$

As a consequence, we have bisimulation and coinduction to our disposal to show equivalence of languages defined in terms of systems of equations involving these additional operators. In order to allow for proofs based on bisimulation-up-to, we first need to extend the congruence closure to deal with intersection and complement. The *boolean congruence closure* $\equiv_R^B$ of a relation R is defined as the least relation $\equiv$ which satisfies the rules of the congruence closure (Definition 4) plus the following two rules:

$$\frac{x_1 \equiv y_1 \qquad x_2 \equiv y_2}{x_1 \wedge x_2 \equiv y_1 \wedge y_2} \qquad \frac{x \equiv y}{\overline{x} \equiv \overline{y}}$$

The associated definition of bisimulation-up-to is as expected. Fortunately, it is also sound as a proof technique for language equality. We do not formally state this here, but in Section 5 we will introduce a general coinduction-up-to theorem which also applies to this case. In the sequel we will simply write $\equiv_R$ for $\equiv_R^B$.

We have already seen that $(\mathcal{P}(A^*), 0, 1, +, \cdot, ^*)$ is a Kleene algebra; it is useful to know that $(\mathcal{P}(A^*), 0, A^*, \overline{-}, +, \wedge)$ is a Boolean algebra. Moreover, below we will need the following property, which holds for any language x and $a \in A$:

$$\overline{xa} = \overline{x}a + \sum_{b \in A \setminus \{a\}} A^* b + 1 \tag{14}$$

For lack of space we do not include a proof; it can very well be shown coinductively.

Example 13. There are unique languages x and y such that

$$x = axa + bxb + a + b + 1 \qquad y = aya + byb + aA^*b + bA^*a$$

x is the language of *palindromes*, i.e., words which are equal to their own reverse. We claim that y must be the language of all *non*-palindromes, i.e., $y = \overline{x}$. We proceed to prove this formally by showing that the relation $R = \{(\overline{x}, y)\}$ is a bisimulation-up-to. The outputs are easily seen to be equal: $o(\overline{x}) = \neg o(x) = \neg o(1) = 0 = o(y)$. We consider the a-derivatives; the b-derivatives are of course similar. In the fourth step we use (14).

$$\begin{aligned} \overline{x}_a &= \overline{x_a} = \overline{xa + 1} = \overline{xa} \wedge \overline{1} = (\overline{x}a + A^*b + 1) \wedge \overline{1} \\ &\equiv_R (ya + A^*b + 1) \wedge \overline{1} = ya \wedge \overline{1} + A^*b \wedge \overline{1} + 1 \wedge \overline{1} = ya + A^*b = y_a \end{aligned}$$

So R is a bisimulation-up-to, proving that y indeed is the complement of x.

5 Bisimulation-up-to for Other Operations

So far we have focused on the regular operations in Section 3 and added complement and intersection in Section 4. One may be interested in other operations on languages as well, such as shuffle or symmetric difference. In this section we provide general conditions under which bisimulation-up-to can be applied.

A *signature* Σ is a collection of operator names $\sigma \in \Sigma$ with associated arities $|\sigma| \in \mathbb{N}$. We define a general congruence closure with respect to a signature:

Definition 14. *For a relation $R \subseteq \mathcal{P}(A^*) \times \mathcal{P}(A^*)$, define the Σ-*congruence closure *$\equiv_R^\Sigma$ of R as the least relation $\equiv$ satisfying the following rules:*

$$\frac{xRy}{x \equiv y} \qquad \frac{}{x \equiv x} \qquad \frac{x \equiv y}{y \equiv x} \qquad \frac{x \equiv y \qquad y \equiv z}{x \equiv z}$$

$$\frac{x_1 \equiv y_1 \qquad \dots \qquad x_n \equiv y_n}{\sigma(x_1, \dots, x_n) \equiv \sigma(y_1, \dots, y_n)} \qquad \textit{for each } \sigma \in \Sigma, n = |\sigma|$$

The congruence closure for the regular operators (Definition 4) and the Boolean congruence closure are special cases of the above definition. Given this generalized congruence closure, we define bisimulation-up-to with respect to a given signature, generalizing Definition 5: a relation $R \subseteq \mathcal{P}(A^*) \times \mathcal{P}(A^*)$ is a *bisimulation-up-to w.r.t. Σ*, if for any $(x, y) \in R$: $o(x) = o(y)$ and for any $a \in A$: $x_a \equiv_R^\Sigma y_a$.

We will give a general condition on the operations involved under which we have an associated coinduction-up-to principle. In order to proceed we define the set of *terms* $T_\Sigma V$ over a signature Σ and a set of variables V by the grammar $t ::= v \mid \sigma(t_1, \dots, t_n)$ where v ranges over V, σ ranges over Σ and $n = |\sigma|$.

Definition 15. *Assume given a signature Σ and for each $\sigma \in \Sigma$ an interpretation $\hat{\sigma} \colon \mathcal{P}(A^*)^n \to \mathcal{P}(A^*)$, where $n = |\sigma|$. We say the semantics of Σ can be given by* behavioural differential equations *if for all $\sigma \in \Sigma$ there is:*

- *a function $f \colon 2^n \to 2$ (n is the arity of σ),*
- *a term $t(v_1, \dots, v_m) \in TV$ for some set of variables V and some $m \in \mathbb{N}$,*
- *for every $a \in A$, a function $g_a \colon \mathcal{P}(A^*)^n \to \mathcal{P}(A^*)^m$ such that for each $i \leq m$ there is $j \leq n$ s.t. either $g_a(x_1, \dots, x_n)(i) = x_j$, or $g_a(x_1, \dots, x_n)(i) = (x_j)_b$ for some $b \in A$, or $g_a(x_1, \dots, x_n)(i) = o(x_j)$,*

such that $o(\hat{\sigma}(x_1, \dots, x_n)) = f(o(x_1), \dots, o(x_n))$ and

$$\hat{\sigma}(x_1, \dots, x_n)_a = t(g_a(\widehat{x_1, \dots}, x_n)) \qquad \textit{for each } a \in A$$

where $\hat{\cdot}$ is the extension of the interpretation of the operators to terms.

Indeed Lemma 2 witnesses that the regular operations can be given by behavioural differential equations, and Lemma 12 shows this additionally for complement and intersection (and see, e.g., [21] for a definition of the shuffle operator by behavioural differential equations). So the following theorem generalizes the coinduction-up-to principle of Theorem 6:

Theorem 16 (Coinduction-up-to). *If the semantics of the operators of a signature Σ can be given by behavioural differential equations, then for any relation R which is a bisimulation-up-to w.r.t Σ: if $(x, y) \in R$ then $x = y$.*

In fact, the coinduction-up-to principle holds even if, in Definition 15, one allows for a different term t_a for every alphabet symbol $a \in A$, instead of a single term t. We have chosen not to include this in the definition for readability reasons.

We conclude this section with an example, adapted from [18], showing that bisimulation-up-to is in general not sound (for operations not given by behavioural differential equations), and as such illustrating the need for a restricted format. Assume, for simplicity, a singleton alphabet $\{a\}$. Consider the constants $\underline{0}$ and $\underline{a}$, and a unary function h, with the following interpretation in languages: $\underline{0} = 0$, $\underline{a} = \{a\}$, $h(0) = 0$ and $h(x) = 1$ for all languages x s.t. $x \neq 0$. The outputs of these two constants and this function are as follows: $o(\underline{0}) = o(\underline{a}) = 0$ and $o(h(x)) = 0$ iff $x = 0$. For the derivatives we have $\underline{a}_a = h(\underline{a})$, $\underline{0}_a = h(\underline{0})$ and $h(x)_a = 0$. Now the relation $R = \{(\underline{0}, \underline{a})\}$ is a bisimulation-up-to, while $\underline{0} \neq \underline{a}$; so for these operations, there is no coinduction-up-to principle. Indeed, h can not be given by behavioural differential equations, because of the case distinction on its argument x rather than that it is based only on the output $o(x)$.

6 Discussion and Related Work

The theory of bisimulation and bisimulation-up-to developed in this paper are instantiations of the general theory of *coalgebras*. Coalgebra [22] is a general mathematical theory for the uniform study of state-based systems including labelled transition systems but also stream systems, various kinds of (weighted or probabilistic) automata, etc. Indeed, *deterministic automata*, as presented in Section 2, are also a certain type of coalgebras. *Bisimulation* is the canonical notion of equivalence of coalgebras, which, for labelled transition systems, coincides with the classical notion introduced by Milner and Park [14,17]. In the case of deterministic automata, the associated instance of bisimulation is precisely the one presented in Section 2. The material of that section is from [21], which contains an extensive investigation of automata and languages as coalgebras.

Bisimulation-up-to classically is a family of enhancements of bisimulation for labelled transition systems [24,18]; it is a rich theory which drastically improves the bisimulation proof method for, e.g., CCS processes. One interesting result based on bisimulation-up-to is the decidability result of [4], on equivalence of context-free processes. Note that these notions of bisimulation-up-to do not directly apply to our case, since our notion of bisimulation for *automata* is different from the classical one for labelled transition systems. Recently the theory of bisimulation-up-to was generalized from labelled transition systems to a large class of coalgebras [20,19], yielding enhancements of the bisimulation proof method for many different kinds of state-based systems. In the present paper we have instantiated this general theory to deterministic automata and certain operations on languages. All operations defined in this paper adhere to specifications in terms of behavioural differential equations [23] (Definition 15). These can easily be represented as so-called *abstract GSOS specifications*, which are a way of defining operational semantics [25]. As a consequence, by the theory of [20,19] bisimulation up to congruence for all of these cases is sound, meaning that any bisimulation-up-to can be extended to a bisimulation.

While we have introduced techniques which are much more widely applicable (as we have shown) than only to regular languages, we proceed to recall some of the related work on checking equivalence of regular expressions. There is a wide range of different tools and techniques tailored towards doing this; we only recall the ones most relevant to our work. CIRC [13] is a general coinductive theorem prover, which can deal with regular expressions. Recently, various algorithms based on Brzozowski derivatives and bisimulations have been implemented in Isabelle [12] and formalized in type theory, yielding an implementation in Coq [6] (while [6] does not mention bisimulations explicitly, their method is based on constructing a bisimulation). An efficient algorithm for deciding equivalence in Kleene algebra, based on automata but not on derivatives and bisimulations, was recently implemented in Coq as well [2]. Of course, one can reason about regular expressions in Kleene algebra; this is however a fundamentally different approach than the coinductive techniques of the present paper. In [9] a proof system for equivalence of regular expressions is presented, based on bisimulations but not on bisimulation-up-to. In [10] a general coinductive axiomatization of regular expression containment is given, based on an interpretation of regular expressions as types. The authors of [10] instantiate their axiomatization with the main coinductive rule from [9]. The focus of [10] is on constructive proofs based on parse trees of regular expressions; instead, we base ourselves on bisimulations between languages. Finally, the recent [1] introduces an efficient algorithm for checking equivalence of non-deterministic automata, based on a different notion of bisimulation up to congruence. One difference with the approach of [1], is that we can deal with (quasi)-equations over arbitrary languages.

If one works with syntactic *terms*, such as regular expressions, rather than with languages, the notion of *bisimulation up to bisimilarity* becomes relevant. In the corresponding proof method, one can relate derivatives, which are then terms, modulo bisimilarity. Since we work directly with languages, in our case this is not necessary; but for dealing with terms our techniques can easily be combined with up-to-bisimilarity (see [20,19]). Bisimulation up to bisimilarity (alone, without context and equivalence closure) was originally introduced in [15], and in the context of automata and languages in [21].

7 Conclusions

We presented a proof method for language equivalence, based on coinduction and bisimulation-up-to. In particular we have considered (quasi-)equations over languages presented by the regular operations of union, concatenation and Kleene star, and additionally by complement and intersection. We have exemplified our approach with novel proofs of classical results.

The presented proof technique is very general, and it applies to undecidable problems such as language equivalence of context-free grammars. Indeed, automation is not the aim of the present paper. Nevertheless, the present techniques can be seen as a foundation for novel interactive theorem provers, and extensions of fully automated tools such as [12,13,6].

References

1. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Proc. POPL (to appear, 2013)
2. Braibant, T., Pous, D.: Deciding Kleene algebras in Coq. Logical Methods in Computer Science 8(1) (2012)
3. Brzozowski, J.: Derivatives of regular expressions. J. ACM 11(4), 481–494 (1964)
4. Christensen, S., Hüttel, H., Stirling, C.: Bisimulation Equivalence is Decidable for All Context-Free Processes. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 138–147. Springer, Heidelberg (1992)
5. Conway, J.: Regular Algebra and Finite Machines. Chapman and Hall (1971)
6. Coquand, T., Siles, V.: A Decision Procedure for Regular Expression Equivalence in Type Theory. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 119–134. Springer, Heidelberg (2011)
7. Foster, S., Struth, G.: Automated Analysis of Regular Algebra. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 271–285. Springer, Heidelberg (2012)
8. Ginsburg, S., Rice, H.: Two families of languages related to ALGOL. J. ACM 9(3), 350–371 (1962)
9. Grabmayer, C.: Using Proofs by Coinduction to Find "Traditional" Proofs. In: Fiadeiro, J.L., Harman, N.A., Roggenbach, M., Rutten, J. (eds.) CALCO 2005. LNCS, vol. 3629, pp. 175–193. Springer, Heidelberg (2005)
10. Henglein, F., Nielsen, L.: Regular expression containment: coinductive axiomatization and computational interpretation. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 385–398. ACM (2011)
11. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. In: LICS, pp. 214–225. IEEE Computer Society (1991)
12. Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. J. Automated Reasoning 49, 95–106 (2012); published online March 2011
13. Lucanu, D., Goriac, E.-I., Caltais, G., Roşu, G.: CIRC: A Behavioral Verification Tool Based on Circular Coinduction. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 433–442. Springer, Heidelberg (2009)
14. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
15. Milner, R.: Calculi for synchrony and asynchrony. TCS 25, 267–310 (1983)
16. Okhotin, A.: Conjunctive and boolean grammars: the true general case of the context-free grammars, `http://users.utu.fi/aleokh/papers/boolean_survey.pdf`
17. 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)
18. Pous, D., Sangiorgi, D.: Enhancements of the bisimulation proof method. In: Advanced Topics in Bisimulation and Coinduction, pp. 233–289. Cambridge University Press (2012)
19. Rot, J., Bonchi, F., Bonsangue, M., Pous, D., Rutten, J., Silva, A.: Enhanced coalgebraic bisimulation, `http://www.liacs.nl/~jrot/up-to.pdf`

20. Rot, J., Bonsangue, M., Rutten, J.: Coalgebraic Bisimulation-Up-To. In: van Emde Boas, P., Groen, F.C.A., Italiano, G.F., Nawrocki, J., Sack, H. (eds.) SOFSEM 2013. LNCS, vol. 7741, pp. 369–381. Springer, Heidelberg (2013)
21. Rutten, J.J.M.M.: Automata and Coinduction (An Exercise in Coalgebra). In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 194–218. Springer, Heidelberg (1998)
22. Rutten, J.: Universal coalgebra: a theory of systems. Theor. Comp. Sci. 249(1), 3–80 (2000)
23. Rutten, J.: Behavioural differential equations: a coinductive calculus of streams, automata, and power series. Theor. Comput. Sci. 308(1-3), 1–53 (2003)
24. Sangiorgi, D.: On the bisimulation proof method. Math. Struct. in Comp. Sci. 8(5), 447–479 (1998), http://dx.doi.org/10.1017/S0960129598002527
25. Turi, D., Plotkin, G.: Towards a mathematical operational semantics. In: LICS, pp. 280–291. IEEE Computer Society (1997)

Ostrowski Numeration and the Local Period of Sturmian Words

Luke Schaeffer

School of Computer Science,
University of Waterloo,
Waterloo, ON N2L 3G1 Canada
`l3schaef@cs.uwaterloo.ca`

Abstract. We show that the local period at position n in a characteristic Sturmian word can be given in terms of the Ostrowski representation for $n + 1$.

Keywords: Sturmian words, Ostrowski representation, local period.

1 Introduction

We consider characteristic Sturmian words, which are infinite words over $\{0, 1\}$ such that the ith character is

$$\lfloor \alpha(i + 1)\rfloor - \lfloor \alpha i\rfloor - \lfloor \alpha\rfloor$$

for some irrational α. We give an alternate definition later better suited to our purposes. Let $f_w(n)$ denote the number of factors of length n in w, also known as the *subword complexity* of w. It is well-known that $f_w(n) = n + 1$ when w is a Sturmian word. On the other hand, the Coven-Hedlund theorem [2] states that $f_w(n)$ is either bounded or $f_w(n) \geq n + 1$ for all n. In this sense, Sturmian words are extremal with respect to subword complexity.

In a recent paper [3], Restivo and Mignosi show that characteristic Sturmian words are also extremal with respect to local period, which we define shortly (as part of Definition 2). Let $p_w(n)$ denote the local period of a word w at position n. The critical factorization theorem states that either $p_w(n)$ is bounded or $p_w(n) \geq n + 1$ for infinitely many n. Restivo and Mignosi show that when w is a characteristic Sturmian word, $p_w(n)$ is at most $n + 1$ and $p_w(n) = n + 1$ infinitely often. Hence, characteristic Sturmian words also have extremal local periods.

Unlike subword complexity, the local period function $p_w(n)$ is erratic. Consider Table 1, which gives the local period at points in F, the Fibonacci word. Although there are patterns in the table (for example, each $p_F(n)$ is a Fibonacci number), it is not obvious how $p_F(n)$ is related to n in general. Shallit [4] showed that $p_F(n)$ is easily computed from the Zeckendorf representation of $n + 1$, and conjectured that for a general characteristic Sturmian word w, $p_w(n)$ is a simple function of the corresponding Ostrowski representation for $n + 1$. In this paper, we confirm Shallit's conjecture by describing $p_w(n)$ in terms of the Ostrowski representation for $n + 1$.

Table 1. The local period function for the Fibonacci word

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$p_F(n)$	1	2	3	1	5	2	2	8	1	3	3	1	13	2	2	5	1	5	2	2	21

2 Notation

Let $\Sigma_2 := \{0,1\}$ for the rest of this paper. We write $w[n]$ to denote the nth letter of a word w (finite or infinite), and $w[i..j]$ for the factor $w[i]w[i{+}1]\cdots w[j{-}1]w[j]$. We use the convention that the first character in w is $w[0]$. When w is finite we let $|w|$ denote its length.

2.1 Repetition Words

Definition 1. *Let w be an infinite word over a finite alphabet Σ. A* repetition word in w at position i *is a non-empty factor $w[i..j]$ such that either $w[i..j]$ is a suffix of $w[0..i-1]$ or $w[0..i-1]$ is a suffix of $w[i..j]$.*

If the infinite word w is recurrent (i.e., every factor in w occurs more than once in w) then every factor occurs infinitely many times. In particular, for every i the prefix $w[0..i-1]$ occurs in $w[i..\infty]$, so there exists a repetition word at every position in a recurrent word.

Definition 2. *Let w be an infinite recurrent word over a finite alphabet Σ. Let $r_w(i)$ denote the shortest repetition word in w at position i. The length of the shortest repetition word, denoted by $p_w(i) := |r_w(i)|$, is called the* local period in w at position i*.*

We note that Sturmian words are recurrent, so $p_w(i)$ and $r_w(i)$ exist at every position for a characteristic Sturmian word w. We omit further discussion of the existence of $p_w(i)$ and $r_w(i)$.

For example, consider the Fibonacci word F shown in Figure 1. The factors $F[5..6] = 01$, $F[5..9] = 01001$ and $F[5..17] = 0100100101001$ are examples of repetition words in the Fibonacci word at position 5. The shortest repetition word at position 5 is $r_F(5) = F[5..6] = 01$ and therefore the local period at position 5 is $p_F(5) = 2$.

3 Characteristic Sturmian Words and the Ostrowski Representation

We define characteristic Sturmian words and the Ostrowski representation based on directive sequences of integers, defined below. For every directive sequence there is a corresponding characteristic Sturmian word. Similarly, for each directive sequence there is an Ostrowski representation associating nonnegative integers with strings.

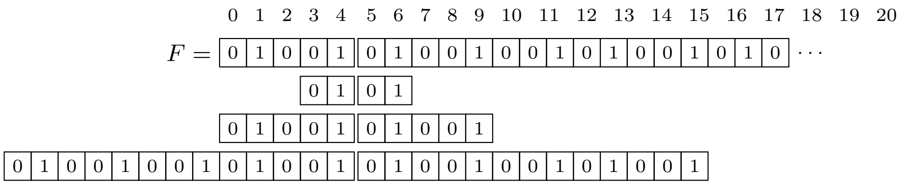


Fig. 1. The Fibonacci word F and some repetition words at position 5

Definition 3. *A* directive sequence $\alpha = \{a_i\}_{i=0}^{\infty}$ *is a sequence of nonnegative integers, where* $a_i > 0$ *for all* $i > 0$.

Directive sequences are in some sense infinite words over the natural numbers, so we use the same indexing/factor notation. The notation $\alpha[i]$ indicates the ith term, a_i. We will frequently separate a directive sequence α into the first term, $\alpha[0]$, and the rest of the sequence, $\alpha[1..\infty]$.

Note that our definitions for characteristic Sturmian words and Ostrowski representations deviate slightly from the definitions given in our references, [5] and [1]. Specifically, there are two main differences between our definition and [5]:

1. We start indexing the directive sequence at zero instead of one.
2. The first term is interpreted differently. For example, if the first term in the directive sequence is a then our characteristic Sturmian word begins with 0^a1, whereas the characteristic Sturmian word in [5] begins with $0^{a-1}1$.

In other words, we are describing the same mathematical objects, but label them with slightly different directive sequences. Any result that does not explicitly reference the terms of the directive sequence will be true for either set of definitions. This includes our main result, Theorem 13.

3.1 Characteristic Sturmian Words

Consider the following collection of morphisms.

Definition 4. *For each* $k \geq 0$*, we define a morphism* $\varphi_k \colon \Sigma_2^* \to \Sigma_2^*$ *such that*

$$\varphi_k(0) = 0^k1$$
$$\varphi_k(1) = 0$$

for all $k \geq 0$.

Given a directive sequence, we use this collection of morphisms to construct a sequence of words.

Definition 5. *Let* α *be a directive sequence. We define a sequence of finite words* $\{X_i\}_{i=0}^{\infty}$ *over* Σ_2 *where*

$$X_n = (\varphi_{\alpha[0]} \circ \cdots \circ \varphi_{\alpha[n-1]})(0).$$

We call $\{X_i\}_{i=0}^{\infty}$ *the* standard sequence*, and we say* X_i *is the* ith characteristic block*.*

Sometimes the characteristic blocks are defined recursively as follows.

Proposition 6. *Let* α *be a directive sequence and let* $\{X_i\}_{i=0}^{\infty}$ *be the corresponding directive sequence. Then*

$$X_n = \begin{cases} 0, & \text{if } n = 0; \\ 0^{\alpha[0]}1, & \text{if } n = 1; \\ X_{n-1}^{\alpha[n-1]} X_{n-2}, & \text{if } n \geq 2. \end{cases}$$

Proof. See Theorem 9.1.8 in [5]. Note that due to a difference in definitions, the authors number the directive sequence starting from one instead of zero, and they treat the first term differently (i.e., they define X_1 as $0^{a_1-1}1$ instead of $0^{\alpha[0]}1$). □

It follows from the proposition that X_{n-1} is a prefix of X_n for each $n \geq 2$, and therefore the limit $\lim_{n\to\infty} X_n$ exists. We define $\mathbf{c}_\alpha$, the characteristic Sturmian word corresponding to the directive sequence α, to be this limit.

$$\mathbf{c}_\alpha := \lim_{n\to\infty} X_n.$$

Then X_n is a prefix of $\mathbf{c}_\alpha$ for each $n \geq 2$.

There is a simple relationship between $\mathbf{c}_\alpha$, $\alpha[0]$ and $\mathbf{c}_{\alpha[1..\infty]}$, given in the following proposition.

Proposition 7. *Let* α *be a directive sequence, and let* $\beta := \alpha[1..\infty]$*. Then*

$$\boldsymbol{c}_\alpha = \varphi_{\alpha[0]}\left(\boldsymbol{c}_\beta\right)$$

Proof. (Sketch) We factor $\varphi_{\alpha[0]}$ out of each X_i and then out of the limit. $\mathbf{c}_\alpha = \lim_{n\to\infty}(\varphi_{\alpha[0]} \circ \cdots \circ \varphi_{\alpha[n-1]})(0) = \varphi_{\alpha[0]}\left(\lim_{n\to\infty}(\varphi_{\alpha[1]} \circ \cdots \circ \varphi_{\alpha[n-1]})(0)\right) = \varphi_{\alpha[0]}\left(\mathbf{c}_\beta\right)$. Alternatively, see Theorem 9.1.8 in [5] for a similar result. □

Notice that if $\alpha[0] = 0$ then $\mathbf{c}_\alpha$ and $\mathbf{c}_\beta$ are the same infinite word up to permutation of the alphabet, since φ_0 swaps 0 and 1. Permuting the alphabet does not affect the local period or repetition words, so henceforth we assume that the first term of any directive sequence is positive (and therefore all terms are positive). Consequently, all characteristic Sturmian words we consider will start with 0 and avoid the factor 11.

Let us give an example of a characteristic Sturmian word. Consider the directive sequence α beginning $1, 3, 2, 2$. Then we can compute the first five terms of the standard sequence

$$\begin{aligned} X_0 &= 0 \\ X_1 &= 01 \\ X_2 &= 0101010 \\ X_3 &= 0101010010101001 \\ X_4 &= 01010100101010010101010010101001010101010. \end{aligned}$$

We know X_4 is a prefix of $\mathbf{c}_\alpha$, so we can deduce the first $|X_4| = 39$ characters of $\mathbf{c}_\alpha$. Thus,

$$\mathbf{c}_\alpha = 010101001010100101010100101010010101010\cdots$$

By Proposition 7, $\mathbf{c}_\alpha$ is equal to $\varphi_1(\mathbf{c}_{\alpha[1..\infty]})$.

$$\begin{aligned}\mathbf{c}_\alpha &= 01\,01\,01\,0\,01\,01\,01\,0\,01\,01\,01\,01\,0\,01\,01\,01\,0\,01\,01\,01\,01\,0\cdots\\ &= \varphi_1(0001000100001000100001\cdots).\end{aligned}$$

3.2 Ostrowski Representation

For each directive sequence α, there is a corresponding characteristic Sturmian word $\mathbf{c}_\alpha$. For each characteristic Sturmian word there is a numeration system, the Ostrowski representation, which is closely related to the standard sequence. For example, if the directive sequence is $\alpha = 1, 1, 1, \ldots$ then $\mathbf{c}_\alpha$ is F, the Fibonacci word. The Ostrowski representation for $\alpha = 1, 1, 1, \ldots$ is the Zeckendorf representation, where we write an integer as a sum of Fibonacci numbers. See chapter three in [5] for a description of these numeration systems, but note that their definition of Ostrowski representation differs from our definition.

Definition 8. *Let α be a directive sequence, and let $\{X_i\}_{i=0}^{\infty}$ be the corresponding standard sequence. Define an integer sequence $\{q_i\}_{i=0}^{\infty}$ where $q_i = |X_i|$ for all $i \geq 0$. Let $n \geq 0$ be an integer. An α-*Ostrowski representation *(or simply* Ostrowski representation *when α is understood) for n is a sequence of non-negative integers $\{d_i\}_{i=0}^{\infty}$ such that*

1. *Only finitely many d_i are nonzero.*
2. *$n = \sum_i d_i q_i$*
3. *$0 \leq d_i \leq \alpha[i]$ for all $i \geq 0$.*
4. *If $d_i = \alpha[i]$ then $d_{i-1} = 0$ for all $i \geq 1$.*

Note that by Proposition 6, we can also generate $\{q_i\}_{i=0}^{\infty}$ directly from α using the following recurrence

$$q_n = \begin{cases} 1, & \text{if } n = 0;\\ \alpha[0] + 1, & \text{if } n = 1;\\ q_{n-1}\alpha[n-1] + q_{n-2}, & \text{if } n \geq 2.\end{cases}$$

It is well-known that for any given directive sequence, there is a unique Ostrowski representation, which we denote $\mathrm{OR}_\alpha(n)$, for every non-negative integer [5]. Also note that formally $\mathrm{OR}_\alpha(n)$ is an infinite sequence $\{d_i\}_{i=0}^{\infty}$, but we often write the terms up to the last nonzero term, e.g., $d_k d_{k-1} \cdots d_1 d_0$, with the understanding that $d_i = 0$ for $i > k$. This is analogous to decimal representation of integers, where we also write the least significant digit last and omit leading zeros.

Theorem 9. *Let α be a directive sequence. Let $n \geq 0$ be an integer, and let $d_k d_{k-1} \cdots d_1 d_0$ be an Ostrowski representation for n. Then*

$$w := X_k^{d_k} X_{k-1}^{d_{k-1}} \cdots X_1^{d_1} X_0^{d_0}$$

is a proper prefix of X_{k+1}, and therefore w is a prefix of $\boldsymbol{c}_\alpha$. Since $|w| = \sum_i d_k |X_i| = n$, it follows that $w = \boldsymbol{c}_\alpha[0..n-1]$.

Proof. This is essentially Theorem 9.1.13 in [5]. □

The following technical lemma relates Ostrowski representations for α and $\alpha[1..\infty]$, in much the same way that Proposition 7 relates $\mathbf{c}_\alpha$ to $\mathbf{c}_{\alpha[1..\infty]}$.

Lemma 10. *Let α be a directive sequence and define $\beta := \alpha[1..\infty]$. Let $n \geq 0$ be an integer with Ostrowski representation $\mathrm{OR}_\alpha(n) = d_k \cdots d_0$. Then there exists an integer $m \geq 0$ such that $\mathrm{OR}_\beta(m) = d_k \cdots d_1$ and*

$$\boldsymbol{c}_\alpha[0..n-1] = \varphi_{\alpha[0]}(\boldsymbol{c}_\beta[0..m-1])0^{d_0}.$$

Furthermore, if $d_0 > 0$ then $\boldsymbol{c}_\beta[m] = 0$.

Proof. We leave it to the reader to show that if $d_k \cdots d_0$ is an α-Ostrowski representation then $d_k \cdots d_1$ is a β-Ostrowski representation, and conversely, if $d_k \cdots d_1$ is a β-Ostrowski representation then $d_k \cdots d_1 0$ is an α-Ostrowski representation. Theorem 9 proves that

$$\mathbf{c}_\alpha[0..n-1] = X_k^{d_k} X_{k-1}^{d_{k-1}} \cdots X_1^{d_1} X_0^{d_0} = \mathbf{c}_\beta[0..m-1]0^{d_0}.$$

Finally, suppose that $d_0 > 0$ and $\mathbf{c}_\beta[m] = 1$ for a contradiction. We consider the integer $n - d_0 + 1$ and its Ostrowski representations. On the one hand, $d_k \cdots d_1 1$ is a valid Ostrowski representation and $d_0 - 1$ less than n. On the other hand,

$$\mathbf{c}_\alpha[0..n-d_0] = \varphi_{\alpha[0]}(\mathbf{c}_\beta[0..m-1])0 = \varphi_{\alpha[0]}(\mathbf{c}_\beta[0..m]),$$

so $\mathrm{OR}_\beta(m+1)$ followed by 0 is another Ostrowski representation for $n - d_0 + 1$. This contradicts the uniqueness of Ostrowski representations. □

Let us continue our earlier example, where we had a directive sequence α beginning $1, 3, 2, 2$. We can compute the first five terms of $\{q_i\}_{i=0}^{\infty}$.

$$\begin{aligned} q_0 &= |X_0| = 1 \\ q_1 &= |X_1| = 2 \\ q_2 &= |X_2| = 7 \\ q_3 &= |X_3| = 16 \\ q_4 &= |X_4| = 39. \end{aligned}$$

In Table 2, we show Ostrowski representations for some small integers. By Theorem 9, we should be able to decompose $\mathbf{c}_\alpha[0..20]$ as $X_3 X_1^2 X_0$ since $\mathrm{OR}_\alpha(21) = 1021$.

$$\begin{aligned} \mathbf{c}_\alpha[0..20] &= 010101001010100101010 \\ &= (0101010010101001)(01)^2 0 \\ &= X_3 X_1^2 X_0. \end{aligned}$$

Table 2. Ostrowski representations where $\alpha = 1, 3, 2, 2, \cdots$

n	$\mathrm{OR}_\alpha(n)$	n	$\mathrm{OR}_\alpha(n)$	n	$\mathrm{OR}_\alpha(n)$	n	$\mathrm{OR}_\alpha(n)$
0	0	15	201	30	1200	45	10030
1	1	16	1000	31	1201	46	10100
2	10	17	1001	32	2000	47	10101
3	11	18	1010	33	2001	48	10110
4	20	19	1011	34	2010	49	10111
5	21	20	1020	35	2011	50	10120
6	30	21	1021	36	2020	51	10121
7	100	22	1030	37	2021	52	10130
8	101	23	1100	38	2030	53	10200
9	110	24	1101	39	10000	54	10201
10	111	25	1110	40	10001	55	11000
11	120	26	1111	41	10010	56	11001
12	121	27	1120	42	10011	57	11010
13	130	28	1121	43	10020	58	11011
14	200	29	1130	44	10021	59	11020

4 Local Periods in Characteristic Sturmian Words

Let α be a directive sequence. Let $p_\alpha(n) := p_{\mathbf{c}_\alpha}(n)$ and $r_\alpha(n) := r_{\mathbf{c}_\alpha}(n)$ be notation for the local period and shortest repetition word for characteristic Sturmian words. In this section we discuss how $p_\alpha(n)$ and $r_\alpha(n)$ are related to $\mathrm{OR}_\alpha(n+1)$.

Definition 11. *Let x, y be words in Σ^*. Then x is a* conjugate *of y if there exist words $u, v \in \Sigma^*$ such that $x = uv$ and $y = vu$.*

Lemma 12. *Let α be a directive sequence, let $\beta := \alpha[1..\infty]$ and $k := \alpha[0]$. Suppose we have integers $m, n \geq 0$ such that $\mathbf{c}_\alpha[0..n] = \varphi_k(\mathbf{c}_\beta[0..m])$. Then*

(i) If u is a repetition word in $\mathbf{c}_\beta$ at position m then there exists a repetition word v in $\mathbf{c}_\alpha$ at position n such that $\varphi_k(u)$ is a conjugate of v.
(ii) If v is a repetition word in $\mathbf{c}_\alpha$ at position n then there exists a repetition word u in $\mathbf{c}_\beta$ at position m such that $\varphi_k(u)$ is a conjugate of v.

In particular, $r_\alpha(n)$ is a conjugate of $\varphi_k(r_\beta(m))$ when $\mathbf{c}_\alpha[0..n] = \varphi_k(\mathbf{c}_\beta[0..m])$.

Proof. We divide the proof into two cases based on whether $\mathbf{c}_\beta[m]$ is 0 or 1. The situation when $\mathbf{c}_\beta[m] = 0$ is shown in Figure 2, and $\mathbf{c}_\beta[m] = 1$ is shown in Figure 3. These figures, along with the more detailed diagrams in Figures 4 and 5 later in the proof, indicate how φ_k maps blocks in $\mathbf{c}_\beta$ to blocks in $\mathbf{c}_\alpha$.

Case $\mathbf{c}_\beta[m] = 0$:
Clearly $\mathbf{c}_\alpha[0..n]$ ends with $0^k1 = \varphi_k(0)$ since $\mathbf{c}_\beta[m] = 0$. This gives us Figure 2.

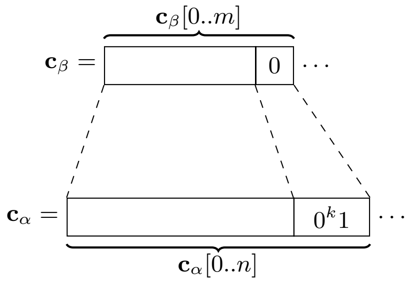


Fig. 2. Simple diagram for $\mathbf{c}_\beta[m] = 0$

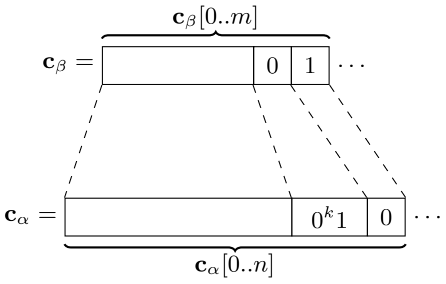


Fig. 3. Simple diagram for $\mathbf{c}_\beta[m] = 1$

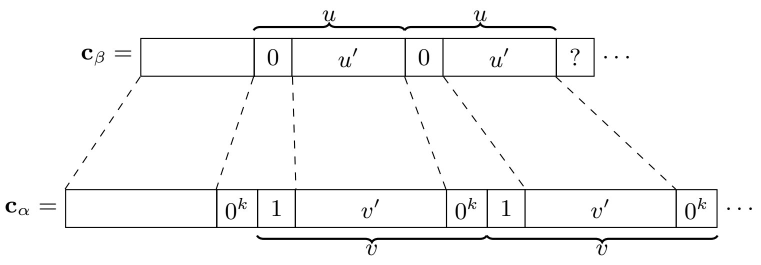


Fig. 4. Detailed diagram for $\mathbf{c}_\beta[m] = 0$

(i) Let u be a repetition word in $\mathbf{c}_\beta$ at position m. If $\mathbf{c}_\beta[0..m-1]$ is a suffix of u then certainly $\mathbf{c}_\alpha[0..n-1] = \varphi_k(\mathbf{c}_\beta[0..m-1])$ is a suffix of $\varphi_k(u)$. Suppose that u is a suffix of $\mathbf{c}_\beta[0..m-1]$. Since $\mathbf{c}_\beta[m] = 0$ we know u begins with 0 and write $u = 0u'$. Since u' is a prefix of $\mathbf{c}_\beta[m+1..\infty]$, we see that $v' := \varphi_k(u')$ is a prefix of $\mathbf{c}_\alpha[n+1..\infty]$. The prefix u' in $\mathbf{c}_\beta[m+1..\infty]$ is followed by 00, 01 or 10. Since $\varphi_k(00)$, $\varphi_k(01)$ and $\varphi_k(10)$ all start with at least k zeros, we deduce that v' (as it occurs at the beginning of $\mathbf{c}_\alpha[n+1..\infty]$) is followed by k zeros. Thus, $v := 1v'0^k$ is a prefix of $\mathbf{c}_\alpha[n..\infty]$. From the other occurrence of u (as a suffix of $\mathbf{c}_\beta[0..m-1]$) we deduce that $1v'0^k$ is also a suffix of $\mathbf{c}_\alpha[0..n-1]$. We conclude that v is a repetition word in $\mathbf{c}_\alpha$ at position, and note that $v = 1v'0^k$ is a conjugate of $0^k1v' = \varphi_k(0u') = \varphi_k(u)$, as required.

(ii) Let v be a repetition word in $\mathbf{c}_\alpha$ at position n. The 1 at position n is preceded by k zeros. Hence, $\mathbf{c}_\alpha[0..n-1]$ ends in 0^k, so v ends in 0^k. Clearly v begins with 1, let v' be such that $v = 1v'0^k$. We do not know whether the trailing 0^k is the beginning of $\varphi_k(0)$ or $\varphi_k(10)$, but in either case v' is $\varphi_k(u')$ for u' a factor of $\mathbf{c}_\beta$.
If $\mathbf{c}_\alpha[0..n-1]$ is a proper suffix of v then $\mathbf{c}_\alpha[0..n-k-1]$ is a suffix of v'. Then $\mathbf{c}_\beta[0..m-1]$ is a suffix of u', and hence $u := 0u'$ is a repetition word in $\mathbf{c}_\beta$ at position m such that v is a conjugate of $\varphi_k(u)$.
Otherwise, v is a suffix of $\mathbf{c}_\alpha[0..n-1]$. The trailing 0^k in this occurrence of v is in the image of $\mathbf{c}_\beta[m] = 0$. The remaining $1v'$ must be preceded by 0^k,

and then $0^k 1v'$ is the image of $0u'$, which occurs as a suffix of $\mathbf{c}_\beta[0..m-1]$. Now we have the situation in Figure 4. It follows that $u := 0u'$ is a repetition word, and $v = 1v'0^k$ is a conjugate of $\varphi_k(u) = 0^k 1v'$.

Case $\mathbf{c}_\beta[m] = 1$:

The characteristic Sturmian words we consider start with 0, so $m \neq 0$. Since $\mathbf{c}_\beta$ does not contain the factor 11, we know $\mathbf{c}_\beta[m-1] = 0$. Therefore $\mathbf{c}_\alpha[0..n]$ ends in $\varphi_k(01) = 0^k 10$, as shown in Figure 3.

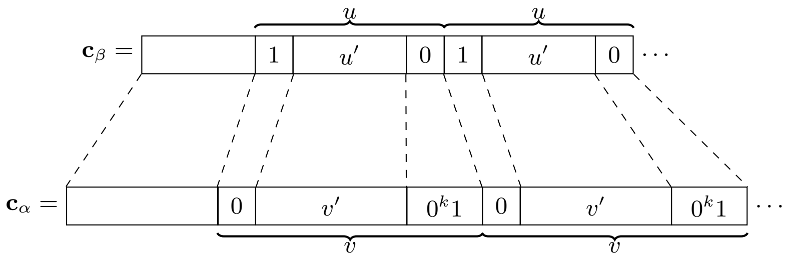


Fig. 5. Detailed diagram for $\mathbf{c}_\beta[m] = 1$

(i) Suppose u is a repetition word in $\mathbf{c}_\beta$ at position m, and let $v := \varphi_k(u)$. We know that $\varphi_k(\mathbf{c}_\beta[0..m-1]) = \mathbf{c}_\alpha[0..n-1]$ and $\varphi_k(\mathbf{c}_\beta[m..\infty]) = \mathbf{c}_\alpha[n..\infty]$. Thus,
- v is a prefix of $\mathbf{c}_\alpha[n..\infty]$ if u is a prefix of $\mathbf{c}_\beta[m..\infty]$
- v is a suffix of $\mathbf{c}_\alpha[0..n-1]$ if u is a suffix of $\mathbf{c}_\beta[0..m-1]$
- $\mathbf{c}_\alpha[0..n-1]$ is a suffix of v if $\mathbf{c}_\beta[0..m-1]$ is a suffix of u.

It follows that v is a repetition word in $\mathbf{c}_\alpha$ at position n.

(ii) Suppose v is a repetition word in $\mathbf{c}_\alpha$ at position n. We know v starts with 0 since $\mathbf{c}_\alpha[n] = 0$, and v ends with 1 since $\mathbf{c}_\alpha[n-1] = 1$, therefore $v = 0v'0^k 1$ for some v'. Then $v' = \varphi_k(u')$ for some u', and we define $u := 1u'0$ so that

$$\varphi_k(u) = \varphi_k(1u'0) = 0v'0^k 1 = v.$$

It is also clear that
- u is a prefix of $\mathbf{c}_\beta[m..\infty]$
- u is a suffix of $\mathbf{c}_\beta[0..m-1]$ if v is a suffix of $\mathbf{c}_\alpha[0..n-1]$
- $\mathbf{c}_\beta[0..m-1]$ is a suffix of u if $\mathbf{c}_\alpha[0..n-1]$ is a suffix of v,

so we conclude that u is a repetition word in $\mathbf{c}_\beta$ at position m.

□

Theorem 13. *Let α be a directive sequence and let $\beta := \alpha[1..\infty]$. Let $n \geq 0$ be a nonnegative integer. Let t be the number of trailing zeros in $\mathrm{OR}_\alpha(n+1)$. Then $r_\alpha(n)$ is a conjugate of X_t, except when all of the following conditions are met:*

- *The last nonzero digit in $\mathrm{OR}_\alpha(n+1)$ is 1.*

– $\mathrm{OR}_\alpha(n+1)$ *contains at least two nonzero digits.*
– *The last two nonzero digits of* $\mathrm{OR}_\alpha(n+1)$ *are separated by an even number of zeros.*

When $\mathrm{OR}_\alpha(n+1)$ *meets these conditions, then* $r_\alpha(n)$ *is a conjugate of* X_{t+1} .

Proof. Let $d_k \cdots d_0 = \mathrm{OR}_\alpha(n+1)$ be the Ostrowski representation of $n+1$. Let t be the number of trailing zeros in $\mathrm{OR}_\alpha(n+1)$. We use induction on t to prove that $r_\alpha(n)$ is a conjugate of X_t, or under the conditions described above, a conjugate of X_{t+1}.

Base case $t=0$: Since $n+1>0$, we have $d_0>0$. By Theorem 9, we have

$$\mathbf{c}_\alpha[0..n] = X_k^{d_k} \cdots X_0^{d_0}.$$

If $d_0 \geq 2$ then we are done since $\mathbf{c}_\alpha[0..n]$ ends in 00. Hence $\mathbf{c}_\alpha[n-1] = \mathbf{c}_\alpha[n] = 0$ and $r_\alpha(n) = 0 = X_0$ is the shortest repetition word at position n. Let us assume without loss of generality that $d_0 = 1$.
According to the induction hypothesis, the second last nonzero digit in $\mathrm{OR}_\alpha(n+1)$ becomes relevant when the last nonzero digit is 1. If d_0 is the only nonzero digit, then $n=0$ and $r_\alpha(0)$ is clearly $\mathbf{c}_\alpha[0]=0$. Otherwise, pick $\ell>0$ minimal such that $d_\ell \neq 0$. That is, let d_ℓ be the second last nonzero digit. Note that by Theorem 9, the word $\mathbf{c}_\alpha[0..n-1]$ ends in X_ℓ.
If ℓ is even then X_ℓ ends in 0 (by a simple induction), so $\mathbf{c}_\alpha[n-1]=0$ and it follows that $r_\alpha(n)=0$. When ℓ is odd, the word X_ℓ ends in X_1 and X_1 ends in 1. It follows that

$$\mathbf{c}_\alpha[0..n-1] = \varphi_{\alpha[0]}(\mathbf{c}_\beta[0..m-1])$$

for some $m \geq 0$. We claim that $\mathbf{c}_\beta[m]=0$, since otherwise

$$\varphi_{\alpha[0]}(\mathbf{c}_\beta[0..m]) = \mathbf{c}_\alpha[0..n]$$

so Lemma 10 states that $\mathrm{OR}_\alpha(n+1)$ ends in 0, contradicting $d_0=1$. Then $\mathbf{c}_\alpha[n..\infty]$ begins with $\varphi_{\alpha[0]}(\mathbf{c}_\alpha[m]) = X_1$, so $r_\alpha(n)=X_1$.

Inductive step $t>0$: We note that removing (or adding) trailing zeros from $\mathrm{OR}_\alpha(n+1)$ does not change whether it satisfies all three conditions in the theorem. We will assume that $\mathrm{OR}_\alpha(n+1)$ does not meet the conditions, since the proof is nearly identical if it does meet the conditions.
Let $\{X_i\}_{i=0}^\infty$ and $\{Y_i\}_{i=0}^\infty$ be standard sequences corresponding to the directive sequences α and β respectively. Lemma 10 states that $\mathbf{c}_\alpha[0..n] = \varphi_{\alpha[0]}(\mathbf{c}_\beta[0..m])$ where $m \geq 0$ is such that

$$\mathrm{OR}_\beta(m+1) = d_k \cdots d_1.$$

Note that $d_k \cdots d_1$ has $t-1$ trailing zeros, so $r_\beta(m)$ is a conjugate of Y_{t-1} by induction. By Lemma 12, $r_\alpha(n)$ is a conjugate of $\varphi_{\alpha[0]}(Y_{t-1}) = X_t$, completing the proof. □

Let us continue our example with $\alpha = 1, 3, 2, 2, \ldots$ as our directive sequence. Recall that

$$\mathbf{c}_\alpha = 01010\,10010\,10100\,10101\,01001\,01010\,01010\,1010\cdots$$

Consider the shortest repetition words at positions 23 through 26. These positions happen to give illustrative examples of the theorem.

$r_\alpha(23) = 0$	$X_0 = 0$	$\mathrm{OR}_\alpha(24) = 1101$
$r_\alpha(24) = 1010100$	$X_2 = 0101010$	$\mathrm{OR}_\alpha(25) = 1110$
$r_\alpha(25) = 01$	$X_1 = 01$	$\mathrm{OR}_\alpha(26) = 1111$
$r_\alpha(26) = 10$	$X_1 = 01$	$\mathrm{OR}_\alpha(27) = 1120$

When $n = 23$, there are no trailing zeros in $\mathrm{OR}_\alpha(24) = 1101$ and we have an odd number of zeros between the last two nonzero digits. Hence, $r_\alpha(23)$ is a conjugate of $X_0 = 0$. Compare this to $n = 25$, where $\mathrm{OR}_\alpha(26) = 1111$ also has no trailing zeros, but the last two ones are adjacent, so $r_\alpha(25)$ is a conjugate of X_1. We are in a similar situation for $n = 24$, but with an trailing zero so $r_\alpha(24)$ is a conjugate of X_2. Finally, consider $n = 26$ where the last two nonzero digits are adjacent and we have a trailing zero, like $n = 24$, but the last nonzero digit is not a one. It follows that $r_\alpha(26)$ is a conjugate of X_1. Although $r_\alpha(25)$ and $r_\alpha(26)$ are both conjugates of X_1, they are not the same.

5 Open Problems and Further Work

It would be interesting to generalize the result to two-sided Sturmian words, with an appropriate definition for local period in two-sided words. We might define a repetition word in $w \in {}^\omega\Sigma^\omega$ at position n as a word that is simultaneously a prefix of $w[n..\infty]$ and a suffix of $w[-\infty..n-1]$. Note that if we extend a characteristic Sturmian word $\mathbf{c}_\alpha$ to a two-sided word w, the local period at position n in w may not be the same as the local period at position n in $\mathbf{c}_\alpha$.

Our main result is about the local period and the shortest repetition word, but Lemma 12 applies to all repetition words at a specific position. Is it possible to extend our result to all repetition words, not just the shortest repetition word? Patterns in the lengths of repetition words for the Fibonacci word suggest that it is possible, but we do not have a specific conjecture.

References

1. Berstel, J., Séébold, P.: Sturmian words. In: Lothaire, M. (ed.) Algebraic Combinatorics on Words. Encyc. of Math. and its Appl., vol. 90, pp. 45–110 (2002)
2. Coven, E., Hedlund, G.: Sequences with minimal block growth. Math. Systems Theory 7, 138–153 (1973)
3. Mignosi, F., Restivo, A.: Characteristic sturmian words are extremal for the critical factorization theorem. Theoret. Comput. Sci. 454, 199–205 (2012)
4. Shallit, J.: Personal Communication (2012)
5. Shallit, J., Allouche, J.P.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press (2003)

Boolean Algebras of Regular ω-Languages

Victor Selivanov and Anton Konovalov

A.P. Ershov Institute of Informatics Systems
Siberian Division Russian Academy of Sciences
Ac. Lavrentyev av. 6, 630090 Novosibirsk, Russia
vseliv@iis.nsk.su

Abstract. We characterize up to isomorphism the Boolean algebras of regular ω-languages and of regular aperiodic ω-languages, and show decidability of classes of languages related to these characterizations.

Keywords: Boolean algebra, Fréchet ideal, regular ω-language, aperiodic ω-language.

1 Introduction

Boolean algebras (BAs) are of principal importance for several branches of mathematics. Accordingly, characterization of naturally arising BA's attracts attention of many researchers. As examples we mention characterizations of natural BAs in logic and computability theory [4,5,12,11,13].

In formal language theory, people consider many classes of languages which form BA's but until recently there was no attempt to characterize those BA's up to isomorphism. To our knowledge, the first papers in this direction are [6,10]. Such characterizations could be of some interest because they provide a new kind information on the classes of languages. Due to Stone duality, this could contribute to understanding of the corresponding Stone spaces which, for the case of regular languages, are closely related to the important and intricate profinite topologies [9,2].

In this note we characterize the Boolean algebras of regular ω-languages and of regular aperiodic ω-languages. These characterizations are similar to the corresponding characterizations for languages of finite words in [10], though some proofs are essentially different. The characterization in [10] turned out closely related to the sparse regular languages [16,8,19]. The same applies to characterizations of this paper (see Corollary 17).

If $\mathbb{B}$ is a BA and α an ordinal, let $F_\alpha(\mathbb{B})$ be the α-th iterated Fréchet ideal of $\mathbb{B}$, $\mathbb{B}^{(\alpha)} = \mathbb{B}/F_\alpha(\mathbb{B})$ is the α-th Fréchet quotient of $\mathbb{B}$ and $\mathbb{B}' = \mathbb{B}^{(1)}$. (See [3] for a detailed treatment, some definitions are recalled in the next section.)

For a finite alphabet Σ, let $\mathbb{R}_\Sigma$ (resp. $\mathbb{A}_\Sigma$) denote the BA of all regular (resp. of all regular aperiodic) languages over Σ. Moreover, let $\mathbb{R}_{\Sigma,\omega}$ (resp. $\mathbb{A}_{\Sigma,\omega}$) denote the BA of all regular (resp. of all regular aperiodic) ω-languages over Σ. These two classes of regular ω-languages are certainly the most important objects in the popular and useful theory of languages of infinite words. The main result of this paper looks as follows:

Theorem 1.

1. *For any alphabet Σ with at least two symbols, $\mathbb{R}_{\Sigma,\omega}$ is an atomic BA with infinitely many atoms, and $\mathbb{R}'_{\Sigma}$ is a countable atomless BA.*
2. *For any alphabet Σ with at least two symbols,*

$$F_0(\mathbb{A}_{\Sigma,\omega}) \subset F_1(\mathbb{A}_{\Sigma,\omega}) \subset \cdots \subset F_\omega(\mathbb{A}_{\Sigma,\omega}) = F_{\omega+1}(\mathbb{A}_{\Sigma,\omega}),$$

for each $n < \omega$ the BA $\mathbb{A}^{(n)}_{\Sigma,\omega}$ is atomic with infinitely many atoms, and $\mathbb{A}^{(\omega)}_{\Sigma,\omega}$ is a countable atomless BA.

From some well-known facts on BA's (see e.g. Chapter 1 of [3]) it follows that items 1,2 characterize the corresponding BA's up to isomorphism. Moreover, our proofs imply decidability of the classes of regular ω-languages related to the main theorem. Note that the case of a unary alphabet (i.e., alphabet with only one symbol) Σ is not interesting for the case of ω-languages because in this case $\mathbb{R}_{\Sigma,\omega}$ and $\mathbb{A}_{\Sigma,\omega}$ are two-element BA's.

As is well known (see e.g. [17]), the class of regular ω-languages (resp. aperiodic ω-languages) coincides with the class of languages satisfying a given sentence of monadic second order (resp. first order) logic of a certain signature. Thus, the main result demonstrates a fundamental difference between the two classes of ω-languages already on the level of the corresponding Tarski-Lindenbaum algebras.

The main result of this paper and the corresponding result about languages of finite words in [10] imply the following

Corollary 2. *For any alphabet Σ with at least two symbols, $\mathbb{R}_{\Sigma,\omega} \simeq \mathbb{R}_{\Sigma}$ and $\mathbb{A}_{\Sigma,\omega} \simeq \mathbb{A}_{\Sigma}$ where $\simeq$ denotes the isomorphism relation.*

Along with the languages of finite words and of infinite words, in automata theory people investigate languages of (jointly) finite and infinite words which are used to specify the behavior of systems which may terminate or not (see e.g. [7]). Our results easily imply the following fact about such languages which we call here $\leq \omega$-languages. Let $\mathbb{R}_{\Sigma,\leq\omega}$ (resp. $\mathbb{A}_{\Sigma,\leq\omega}$) denote the BA of all regular (resp. of all regular aperiodic) $\leq \omega$-languages over Σ.

Corollary 3. *For any alphabet Σ with at least two symbols, $\mathbb{R}_{\Sigma,\leq\omega} \simeq \mathbb{R}_{\Sigma,\omega}$ and $\mathbb{A}_{\Sigma,\leq\omega} \simeq \mathbb{A}_{\Sigma,\omega}$ where $\simeq$ denotes the isomorphism relation.*

To prove the main result, we establish several other facts about regular ω-languages which might be interesting in their own right. For instance, in Theorem 7 we show that, given a Muller automaton, one can effectively compute the cardinality of the corresponding regular ω-language. To our knowledge, this fact is new, although its important particular case (the decidability of the class of finite ω-languages) is well known.

In Sections 2 and 3 we provide the necessary background on BAs and on regular languages, respectively. In Sections 4 and 5 we characterize the BAs $\mathbb{R}_{\Sigma,\omega}$ and $\mathbb{A}_{\Sigma,\omega}$, respectively.

2 Preliminaries on Boolean Algebras

In this section we very briefly recall some notions and facts on BA's used in the sequel (more details are contained in [10]). We assume the reader to be familiar with basic notions related to BA's like ideal of a BA, quotient-algebra of a BA modulo a given ideal, and canonical homomorphism of a BA onto its quotient-algebra (for detailed treatments of BA's we refer to [3,14]). BAs are considered in the signature $\{\cup, \cap, \bar{}, 0, 1\}$. Any BA carries the induced partial order $\leq$ such that $x \leq y$ iff $x \cap y = x$.

Recall that element a of a BA an *atom* if $a \neq 0$ and $x < a$ implies $x = 0$. A BA $\mathbb{B}$ is *atomless* if it has no atom, and it is *atomic* if below any non-zero element there is an atom. The ideal of a BA $\mathbb{B}$ generated by atoms is called *Fréchet ideal* of $\mathbb{B}$. It consists of all finite unions (including the empty union which coincides with 0) of atoms and it is denoted by $F(\mathbb{B})$. The quotient-algebra $\mathbb{B}/F(\mathbb{B})$ is called the *Fréchet quotient* of $\mathbb{B}$ denoted also by $\mathbb{A}'$.

Define the transfinite sequence $\{F_\alpha(\mathbb{B})\}$ of *iterated Fréchet ideals* of a BA $\mathbb{B}$ as follows:

$$F_0(\mathbb{B}) = \{0\},\ F_{\beta+1}(\mathbb{B}) = \{x \mid x/F_\beta(\mathbb{B}) \in F(\mathbb{B}^{(\beta)})\}$$

where $\mathbb{B}^{(\beta)} = \mathbb{B}/F_\beta(\mathbb{B})$, and $F_\alpha(\mathbb{B}) = \bigcup_{\beta<\alpha} F_\beta(\mathbb{B})$ for a limit ordinal α. This sequence is ascending under inclusion, and $F_\alpha(\mathbb{B}) = F_{\alpha+1}(\mathbb{B})$ for some α; the least ordinal α with this property is called the *ordinal type of* $\mathbb{B}$ and is denoted $\sigma(\mathbb{B})$. In this paper we use Fréchet ideals only for $\alpha \leq \omega$.

3 Preliminaries on Regular Languages

Here we briefly recall some notation, notions and facts on regular languages used in the sequel. For a systematic treatment see e.g. [15,7,17,19].

Let Σ^* and Σ^ω denote respectively the sets of all words and of all ω-words (i.e. sequences $\alpha : \omega \to \Sigma$) over a finite alphabet Σ. The empty word is denoted by ε. Let $\Sigma^{\leq\omega} = \Sigma^* \cup \Sigma^\omega$ and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. We use some almost standard notation concerning words and ω-words, so we are not pedantic in reminding it here. For $w \in \Sigma^*$ and $\xi \in \Sigma^{\leq\omega}$, $w \sqsubseteq \xi$ means that w is a prefix of ξ, $w \cdot \xi = w\xi$ denote the concatenation, $l = |w|$ is the length of $w = w(0) \cdots w(l-1)$. For $w \in \Sigma^*, W \subseteq \Sigma^*$ and $A \subseteq \Sigma^{\leq\omega}$, let $w \cdot A = \{w\xi : \xi \in A\}$ and $W \cdot A = \{w\xi : w \in W, \xi \in A\}$. For $u \in \Sigma^+$, u^ω denotes the infinite concatenation $uu \cdots \in \Sigma^\omega$. For each $W \subseteq \Sigma^+$ let $W^\omega = \{w_0 w_1 \cdots \mid w_i \in W\}$. Sometimes we use some other standard notation related to regular expressions.

By an *automaton* (over Σ) we mean a triple $\mathcal{M} = (Q, f, q_0)$ consisting of a finite non-empty set Q of states, a transition function $f : Q \times \Sigma \to Q$ and an initial state $q_0 \in Q$. The transition function is naturally extended to the function $f : Q \times \Sigma^* \to Q$ defined by induction $f(q, \varepsilon) = q$ and $f(q, u \cdot x) = f(f(q, u), x)$, where $u \in \Sigma^*$ and $x \in X$. We often simplify $f(q, u)$ to $q \cdot u$.

Similarly, we may define the function $f : Q \times \Sigma^\omega \to Q^\omega$ by $f(q, \xi)(n) = f(q, \xi[n])$ where $\xi[n] = \xi(0) \cdots \xi(n-1)$ is the prefix of ξ of length $n < \omega$. Relate to any automaton $\mathcal{M}$ the set of *Büchi macrostates* $C_\mathcal{M} = \{f_\mathcal{M}(\xi) \mid \xi \in \Sigma^\omega\}$

where $f_{\mathcal{M}}(\xi)$ is the set of states which occur infinitely often in the sequence $f(q_0, \xi) \in Q^\omega$. Note that in this paper we consider only deterministic finite automata such that any state is reachable from q_0.

Automata equipped with appropriate additional structures may be used as acceptors (devices accepting words or ω-words). A *word acceptor* has the form $(\mathcal{M}, F)$ where $\mathcal{M}$ is an automaton and $F \subseteq Q$; it recognizes the set $L(\mathcal{M}, F) = \{u \in \Sigma^* \mid q_0 \cdot u \in F\}$. It is well known that word acceptors recognize exactly the regular languages. The class $\mathcal{R}_\Sigma$ of regular languages is closed under the Boolean operations.

A *Muller acceptor* has the form $(\mathcal{M}, \mathcal{F})$ where $\mathcal{M}$ is an automaton and $\mathcal{F} \subseteq C_{\mathcal{M}}$; it recognizes the set $L_\omega(\mathcal{M}, \mathcal{F}) = \{\xi \in \Sigma^\omega \mid f_{\mathcal{M}}(\xi) \in \mathcal{F}\}$. It is well known that Muller acceptors recognize exactly the *regular ω-languages* called also regular sets in this paper. The class $\mathcal{R}_{\Sigma,\omega}$ of regular ω-languages is closed under the Boolean operations. It coincides with the class of finite unions of sets $U \cdot W^\omega$ where U, W are regular languages.

For $q \in Q$ and $u \in \Sigma^*$, let (q, u) denote the path in $\mathcal{A}$ along u started at q, and let $Q(q, u) = \{q \cdot v \mid v \sqsubseteq u\}$. We say that a path (q, u) *meets* a set of states $G \subseteq Q$ if $Q(q, u) \cap G \neq \emptyset$. A path (q, u) is a *cycle of* $\mathcal{M}$ if u is nonempty and $q \cdot u = q$. The cycle is *simple* if it has no repeated vertices other that the starting and ending vertices. Note that the set $C_{\mathcal{M}}$ of Büchi macrostates coincides with the set of all $Q(q, u)$ where (q, u) is a cycle of $\mathcal{M}$.

Define the preorder $\leq$ on $C_{\mathcal{M}}$ as follows: $G \leq H$ if there is a path of $\mathcal{M}$ starting in G and ending in H. Let $\equiv$ denote the equivalence relation on $C_{\mathcal{M}}$ induced by $\leq$. Note that if $G \equiv H$ then $K \equiv G$ for some $K \supseteq G \cup H$, i.e. any element $[G]$ of the quotient-set $C_{\mathcal{M}}/\equiv$ has a greatest set under inclusion; these greatest sets are called *strongly connected components* (SCC's) of $\mathcal{M}$ and they may serve as canonical representatives for the equivalence classes $[G]$.

We recall the following classical facts of automata theory. An automaton $\mathcal{M}$ is called *counter-free* if $q \cdot u^n = q$ implies $q \cdot u = q$, for all $q \in Q$, $u \in \Sigma^+$ and $n > 0$.

Proposition 4. *For any $L \in \mathcal{R}_\Sigma$ the following conditions are equivalent:*

1. *There is $n < \omega$ such that $xy^n z \in L$ iff $xy^{n+1} z \in L$ for all $x, y, z \in \Sigma^*$.*
2. *L is recognized by a counter-free word acceptor.*

Languages satisfying the conditions in the last proposition are called *aperiodic.* By *aperiodicity index* of an aperiodic language L we mean the least number n satisfying the condition 1 above. The class $\mathcal{A}_\Sigma$ of regular aperiodic languages is closed under the Boolean operations.

Proposition 5. *For any $L \in \mathcal{R}_{\Sigma,\omega}$ the following conditions are equivalent:*

1. *There is $n < \omega$ such that $xu^n yz^\omega \in L$ iff $xu^{n+1} yz^\omega \in L$, and $x(yu^n z)^\omega \in L$ iff $x(yu^{n+1} z)^\omega \in L$ for all $x, u, y, z \in \Sigma^*$.*
2. *L is a finite union of sets UW_i^ω where $U \subseteq \Sigma^*$ and $\subseteq \Sigma^+$ are regular aperiodic languages.*

3. L is recognized by a counter-free Muller acceptor.

Languages satisfying the conditions in the last proposition are called *regular aperiodic ω-languages.* By *aperiodicity index* of a regular aperiodic ω-language L we mean the least number n satisfying the condition 1 above. The class $\mathcal{A}_{\Sigma,\omega}$ of regular aperiodic ω-languages is closed under the Boolean operations.

Finally, we recall [7] that an $\leq\omega$-language $L \subseteq \Sigma^{\leq\omega}$ is regular (resp. regular aperiodic) iff $L \cap \Sigma^*$ is a regular (resp. regular aperiodic) language and $L \cap \Sigma^\omega$ is a regular (resp. regular aperiodic) ω-language. This immediately implies that $\mathbb{R}_{\Sigma,\leq\omega} \simeq \mathbb{R}_\Sigma \times \mathbb{R}_{\Sigma,\omega}$ and $\mathbb{A}_{\Sigma,\leq\omega} \simeq \mathbb{A}_\Sigma \times \mathbb{A}_{\Sigma,\omega}$.

4 Boolean Algebra $\mathbb{R}_{\Sigma,\omega}$

First we characterize Muller automata $(\mathcal{M},\mathcal{F})$ such that the corresponding set $L_\omega(\mathcal{M},\mathcal{F})$ has a given cardinality $\alpha \in \omega \cup \{\omega, 2^\omega\}$. Details of this characterization will be used in subsequent characterizations of Boolean algebras.

We call a Büchi macrostate F of an automaton $\mathcal{M}$ *branching* if there are at least two distinct loops marked by $\sqsubseteq$-incomparable words and contained in the macrostate. More formally and equivalently, there are incomparable words $v, w \in \Sigma^*$ and a state $q \in F$ such that $q \cdot v = q = q \cdot w$ and $Q(q,v) = F = Q(q,w)$.

Proposition 6. *Let F be a Büchi macrostate of an automaton $\mathcal{M}$. If F is branching then $|L_\omega(\mathcal{M},\{F\})| = 2^\omega$, otherwise $L_\omega(\mathcal{M},\{F\}) = L(\mathcal{M},\{q\}) \cdot u^\omega$ for some $q \in F$ and $u \in \Sigma^+$.*

Proof. Let F be branching via q, v, w. Choose $x \in \Sigma^*$ with $q_0 \cdot x = q$, and relate to any $C \subseteq \omega$ the ω-word $\xi_C = xy_0y_1\cdots$ where $y_n = v$ for $n \in C$ and $y_n = w$ for $n \in \omega \setminus C$. Clearly, $\xi_C \in L_\omega(\mathcal{M},\{F\})$ for each $C \subseteq \omega$, hence it suffices to check that $\xi_C \neq \xi_B$ for all distinct $C, B \subseteq \omega$. Let n be the smallest number in $(C \setminus B) \cup (B \setminus C)$. Then

$$\{\xi_C, \xi_B\} = \{xy_0 \cdots y_{n-1}v\eta, xy_0 \cdots y_{n-1}w\zeta\}$$

for some $\eta, \zeta \in \Sigma^\omega$. Since v, w are $\sqsubseteq$-incomparable, $\xi_C \neq \xi_B$.

Let now F be non-branching. Choose an arbitrary $q \in F$ and a shortest word $u \in \Sigma^+$ such that $q \cdot u = q$ and $Q(q,u) = F$. We claim that $L_\omega(\mathcal{M},\{F\}) = L(\mathcal{M},\{q\}) \cdot u^\omega$. The inclusion from right to left is obvious. For the converse, let $\xi \in L_\omega(\mathcal{M},\{F\})$, then $\xi = x\zeta$ for some $x \in \Sigma^*$ with $q_0 \cdot x = q$ and $\zeta \in \Sigma^\omega$ with $Q(q,\zeta) = F = f_\mathcal{M}(\xi)$. Then $\zeta = v_0\eta_0$ for some $v_0 \in \Sigma^+$ with $q \cdot v_0 = q$ and $Q(q,v_0) = F$. Since u is shortest and F is non-branching, $u \sqsubseteq v_0$, hence $\zeta = u\zeta_1$ for some $\zeta_1 \in \Sigma^\omega$ with $Q(q,\zeta_1) = F = f_\mathcal{M}(\xi)$. Then $\zeta_1 = v_1\eta_1$ for some $v_1 \in \Sigma^+$ with $q \cdot v_1 = q$ and $Q(q,v_1) = F$. Since u is shortest and F is non-branching, $u \sqsubseteq v_1$ hence $\zeta_1 = u\zeta_2$ for some $\zeta_2 \in \Sigma^\omega$ with $Q(q,\zeta_2) = F = f_\mathcal{M}(\xi)$. Continuing in this manner, we obtain $\xi = xu^\omega \in L(\mathcal{M},\{q\}) \cdot u^\omega$. □

The next result might be of interest in its own right.

Theorem 7. *Let $(\mathcal{M},\mathcal{F})$ be a Muller acceptor.*

1. *If there is a branching macrostate $F \in \mathcal{F}$ then $|L_\omega(\mathcal{M}, \mathcal{F})| = 2^\omega$, otherwise $|L_\omega(\mathcal{M}, \mathcal{F})| \leq \omega$.*
2. *If all $F \in \mathcal{F}$ are non-branching and there is $F \in \mathcal{F}$ such that the SCC $[F]$ is non-singleton then $|L_\omega(\mathcal{M}, \mathcal{F})| = \omega$.*
3. *If all $F \in \mathcal{F}$ are non-branching with singleton SCC's $[F]$ and there is a path from q_0 to some $F \in \mathcal{F}$ that meets some Büchi macrostate $G \neq F$ then $|L_\omega(\mathcal{M}, \mathcal{F})| = \omega$.*
4. *If all conditions in 1 — 3 are false then $|L_\omega(\mathcal{M}, \mathcal{F})| < \omega$.*
5. *The cardinality $|L_\omega(\mathcal{M}, \mathcal{F})|$ is computable.*

Proof. 1. Follows from Proposition 6.

2. By item 1, $|L_\omega(\mathcal{M}, \mathcal{F})| \leq \omega$. It suffices to check that there are $x \in \Sigma^*$ and $y, u, v \in \Sigma^+$ such that $xv^*yu^\omega \subseteq L_\omega(\mathcal{M}, \mathcal{F})$ and y, v are $\sqsubseteq$-incomparable, because then $xv^ktyu^\omega \neq xv^lyu^\omega$ for $k \neq l$.

Let F, G be distinct Büchi macrostates such that $F \equiv G$ and $F \in \mathcal{F}$. Assume first that $F \not\subseteq G$, so there is $q \in F \setminus G$. Let u be the shortest word from the previous proof. Let r be an arbitrary element of G. Since $F \equiv G$, $r \cdot y = q$ and $q \cdot z = r$ for some words y, z. Let w be a word satisfying $r \cdot w = r$ and $Q(r, w) = G$. Since $q \in F \setminus G$, the words y, w are $\sqsubseteq$-incomparable. Finally, let $v = wyz$ and let x be a word satisfying $q_0 \cdot x = r$. Then x, y, u, v have the desired properties.

It remains to consider the case $F \subset G$. Let $q \in F$ and u be as in the previous proof. and let v satisfy $q \cdot v = q$ and $Q(q, v) = G$. Clearly, v can be chosen to be $\sqsubseteq$-incomparable with u. Finally, let $y = u$ and let x be a word satisfying $q_0 \cdot x = q$. Then x, y, u, v have the desired properties.

3. Let $F \in \mathcal{F}$ and G be Büchi macrostates with the specified properties. Since $[F]$ is singleton, $F \cap G = \emptyset$. Let $q \in F$ and u be as in the previous proof. Let $r \in G$ and let v satisfy $q \cdot v = q$ and $Q(q, v) = G$. Let y be a word with $r \cdot y = q$, then y and v are $\sqsubseteq$-incomparable. Finally, let x be a word satisfying $q_0 \cdot x = r$. Then x, y, u, v satisfy the condition in the proof of item 2, hence that proof applies.

4. For any $F \in \mathcal{F}$, choose $q_F \in F$ and u_F as in the proof of Proposition 6. Then $L_\omega(\mathcal{M}, \mathcal{F}) = \bigcup\{L(\mathcal{M}, \{q_F\}) \cdot u_F^\omega \mid F \in \mathcal{F}\}$. Since any path from q_0 to q_F meets no Büchi macrostate distinct from F, all languages $L_\omega(\mathcal{M}, \{F\})$, $F \in \mathcal{F}$, are finite, hence $L_\omega(\mathcal{M}, \mathcal{F})$ is finite.

5. Given a Muller acceptor $(\mathcal{M}, \mathcal{F})$, one can compute which of the alternative conditions in 1 — 4 holds. In cases 1 — 3 this returns $|L_\omega(\mathcal{M}, \mathcal{F})|$. In case 4 one has additionally to count the cardinalities of the pairwise disjoint languages $L(\mathcal{M}, \{q_F\}) \cdot u_F^\omega$ which is clearly computable. □

Corollary 8. *Let $L \in \mathcal{R}_{\Sigma,\omega}$ be infinite. Then there are $x \in \Sigma^*$ and $y, u, v \in \Sigma^+$ such that $xv^*yu^\omega \subseteq L$ and y, v are $\sqsubseteq$-incomparable.*

Proof. Choose a Muller acceptor $(\mathcal{M}, \mathcal{F})$ recognizing L. By Theorem 7, $(\mathcal{M}, \mathcal{F})$ satisfies exactly one of the condition in items 1 — 3. By the proof of that proposition, in any in these cases there are words x, y, u, v with the desired properties. □

We are now ready to prove item 1 of the main theorem from Introduction.

Theorem 9. *For any alphabet Σ with at least two symbols, $\mathbb{R}_{\Sigma,\omega}$ is an atomic BA with infinitely many atoms, and $\mathbb{R}'_{\Sigma}$ is a countable atomless BA.*

Proof. Obviously, the atoms of $\mathbb{R}_{\Sigma,\omega}$ are exactly the singleton ω-languages of the form $\{xu^\omega\}$, hence the first assertion holds. For the second assertion, it suffices to show that for any infinite set $L \in \mathbb{R}_{\Sigma,\omega}$ there is an infinite regular ω-language $M \subseteq L$ such that $L \setminus M$ is infinite. By Corollary 8, there are $x \in \Sigma^*$ and $y, u, v \in \Sigma^+$ such that $xv^*yu^\omega \subseteq L$ and y, v are $\sqsubseteq$-incomparable. Then the regular ω-language $M = x(vv)^*yu^\omega$ has the desired properties. □

As mentioned in the Introduction, Theorem 9 characterizes the BA $\mathbb{R}_\Sigma$ up to isomorphism. This implies the following fact:

Corollary 10. *For any alphabet Σ, $\mathbb{R}_{\Sigma,\leq\omega} \simeq \mathbb{R}_\Sigma$.*

Proof. Considering the cases $|\Sigma| \geq 2$ and $|\Sigma| = 1$ and using the remark $\mathbb{R}_{\Sigma,\leq\omega} \simeq \mathbb{R}_\Sigma \times \mathbb{R}_{\Sigma,\omega}$ from the end of Section 3 one easily checks that BA $\mathbb{R}_{\Sigma,\leq\omega}$ has the properties specified in Theorem 9. Since BA $\mathbb{R}_\Sigma$ has the same properties (see [6,10]), $\mathbb{R}_{\Sigma,\leq\omega} \simeq \mathbb{R}_\Sigma$. □

5 Boolean Algebra $\mathbb{A}_{\Sigma,\omega}$

For the sequel we need the following lemma which follows easily from Proposition 4 and Theorem 2.1 in [1]. A non-empty word $u \in \Sigma^+$ is called *primitive* if, for each $v \in \Sigma^+$, $v^n = u$ implies $n = 1$ (and hence $v = u$).

Lemma 11. *For any $u \in \Sigma^*$, $u^* \in \mathcal{A}_\Sigma$ iff u is either empty or primitive. For any $u \in \Sigma^+$, $u^\omega \in \mathcal{A}_{\Sigma,\omega}$.*

The next two lemmas clarify the structure of Fréchet ideals of $\mathbb{A}_{\Sigma,\omega}$.

Lemma 12. *For any $k < \omega$ and $x, y_i, z_i \in \Sigma^*$ with $y_1^*, \cdots, y_k^* \in \mathcal{A}_\Sigma$ and $u \in \Sigma^+$, the element $xy_1^*z_1 \cdots y_k^*z_ku^\omega / F_k(\mathbb{A}_{\Sigma,\omega})$ is either zero or an atom of the BA $\mathbb{A}^{(k)}_{\Sigma,\omega}$. If L is a finite union of such languages $xy_1^*z_1 \cdots y_k^*z_ku^\omega$ then $L \in F_{k+1}(\mathbb{A}_{\Sigma,\omega})$.*

Proof. Since the second assertion follows from the first one, it suffices to check the first assertion by induction on k. For $k = 0$ the assertion is obvious because $\{xu^\omega\}$ is an atom of $\mathbb{A}_{\Sigma,\omega} = \mathbb{A}^{(0)}_{\Sigma,\omega}$.

Let $k = 1$. The case $y_1 = \varepsilon$ is trivial, so assume y_1 to be non-empty (hence, primitive by Lemma 11). Then $L = xy_1^*z_1u^\omega$ is infinite and we have to show that if $M \subseteq L$ is an infinite regular ω-aperiodic language then $L \setminus M$ is finite. We have $xy_1^mz_1u^\omega \in M$ for infinitely many m. By Proposition 5, $xy_1^mz_1u^\omega \in M$ for all $m \geq n$ where n is the aperiodicity index of M. Therefore, $L \setminus M$ is finite.

Let now $k \geq 2$. For any $n < \omega$, set $L_n = xy_1^ny_1^*z_1 \cdots y_k^ny_k^*z_ku^\omega$, then $L = L_0 \supset L_1 \supset L_2 \supset \cdots$ where $L = xy_1^*z_1 \cdots y_k^*z_ku^\omega$. It suffices to show that for any regular aperiodic ω-language $M \subseteq L$ at least one of ω-languages $M, L \setminus M$ is in $F_k(\mathbb{A}_\Sigma)$. We distinguish two cases.

Case 1. $L_n \subseteq \overline{M}$ for some n. Then $M = L \setminus \overline{M} \subseteq L \setminus L_n \in F_k(\mathbb{A}_{\Sigma,\omega})$ by induction on k. Thus, $M \in F_k(\mathbb{A}_{\Sigma,\omega})$.

Case 2. $L_n \not\subseteq \overline{M}$ for all n, i.e. for any n there are $m_1, \ldots, m_k \geq n$ such that $xy_1^{m_1}z_1 \cdots y_k^{m_k}z_k u^\omega \in M$. By Proposition 5, $xy_1^{m_1}z_1 \cdots y_k^{m_k}z_k u^\omega \in M$ for all $m_1, \ldots, m_k \geq m$ where m is the aperiodicity index of M. Then $L_m \subseteq M$, hence $L \setminus M \subseteq L \setminus L_m \in F_k(\mathbb{A}_{\Sigma,\omega})$, hence $L \setminus M \in F_k(\mathbb{A}_{\Sigma,\omega})$. □

Lemma 13. *Let $x, y \in \Sigma^*$ and $u, v_1, v_2 \in \Sigma^+$ be such that the words u, v_1, v_2 are primitive and v_1, v_2 are $\sqsubseteq$-incomparable. Then $xv_1^*v_2yu^\omega \notin F_1(\mathbb{A}_{\Sigma,\omega})$,*

$$xv_1^*v_2v_1^*v_2yu^\omega \notin F_2(\mathbb{A}_{\Sigma,\omega}),\ xv_1^*v_2v_1^*v_2v_1^*v_2yu^\omega \notin F_3(\mathbb{A}_{\Sigma,\omega})$$

and so on.

Proof. The assertion $xv_1^*v_2yu^\omega \notin F_1(\mathbb{A}_{\Sigma,\omega})$ is clear because the set $xv_1^*v_2yu^\omega$ is infinite while $F_1(\mathbb{A}_{\Sigma,\omega})$ is the class of finite regular ω-languages.

The set $xv_1^*v_2v_1^*v_2yu^\omega$ is a disjoint union of sets $K_n = xv_1^n v_2v_1^*v_2yu^\omega$, and, for each n, $K_n/F_1(\mathbb{A}_{\Sigma,\omega})$ is an atom of $\mathbb{A}'_\Sigma$ by the previous lemma. Therefore, $xv_1^*v_2v_1^*v_2yu^\omega \notin F_2(\mathbb{A}_{\Sigma,\omega})$. Continuing in this manner, we derive the desired assertions. □

We say that a Muller acceptor $(\mathcal{M}, \mathcal{F})$ *has an ω-pattern* (cf. with the corresponding notion in [10]) if there are $x, y \in \Sigma^*$ and $u, v_1, v_2 \in \Sigma^+$ such that v_1, v_2 *are* $\sqsubseteq$-incomparable, $q_0 \cdot x = q_0 \cdot xv_1 = q_0 \cdot xv_2$, $q_0 \cdot xyu = q_0 \cdot xy$, and $Q(q_0 \cdot xy, u) \in \mathcal{F}$.

Lemma 14. *For any counter-free Muller acceptor $(\mathcal{M}, \mathcal{F})$, if $L_\omega(\mathcal{M}, \mathcal{F})$ is in $F_\omega(\mathbb{A}_\Sigma)$ then $(\mathcal{M}, \mathcal{F})$ has no ω-pattern.*

Proof. By contraposition, suppose that $(\mathcal{A}, \mathcal{F})$ has an ω-pattern with some words x, y, u, v_1, v_2 as above. By Proposition 5, the words v_1, v_2 are primitive. We have to show that $L \notin F_\omega(\mathbb{A}_\Sigma)$. Since $x(v_1 + v_2)^*yu^\omega \subseteq L_\omega(\mathcal{A}, \mathcal{F})$, it suffices to show that $xv_1^*v_2yu^\omega \notin F_1(\mathbb{A}_{\Sigma,\omega})$,

$$xv_1^*v_2v_1^*v_2yu^\omega \notin F_2(\mathbb{A}_{\Sigma,\omega}),\ xv_1^*v_2v_1^*v_2v_1^*v_2yu^\omega \notin F_3(\mathbb{A}_{\Sigma,\omega})$$

and so on. But this holds by the previous lemma. □

We are ready to provide useful characterizations of the iterated Fréchet ideals of $\mathbb{A}_{\Sigma,\omega}$.

Theorem 15. *For any $L \in \mathcal{A}_{\Sigma,\omega}$ the following conditions are equivalent:*

1. *$L \in F_\omega(\mathbb{A}_{\Sigma,\omega})$.*
2. *L is recognized by a counter-free Muller acceptor that has no ω-pattern.*
3. *L is a finite union of sets $xy_0^*z_0 \cdots y_k^*z_k u^\omega$ where $k < \omega$, $x, y_i, z_i \in \Sigma^*$, $u \in \Sigma^+$ and $y_0^*, \cdots, y_k^* \in \mathcal{A}_\Sigma$.*

Proof. 1→2. Let $(\mathcal{M}, \mathcal{F})$ be a counter-free Muller acceptor recognizing L. By the previous lemma, $(\mathcal{M}, \mathcal{F})$ has no ω-pattern.

2→3. The acceptor $(\mathcal{M}, \mathcal{F})$ above cannot satisfy a condition of items 1,2 of Proposition 7 because any of those conditions induces an ω-pattern for $(\mathcal{M}, \mathcal{F})$. Hence, $(\mathcal{M}, \mathcal{F})$ satisfies the condition of either 3 or 4. In the last case L is finite and we are done, so let $(\mathcal{M}, \mathcal{F})$ satisfy the condition of 3. For any Büchi macrostate G from that condition the SCC $[G]$ is singleton because otherwise $(\mathcal{M}, \mathcal{F})$ has an ω-pattern.

Consider now all chains of Büchi macrostates $F_1 < \cdots < F_m < F \in \mathcal{F}$ of maximal possible lengths $m \geq 0$. As in Lemma 1 of [10], there exist $x \in \Sigma^*$, $u, y_i, z_i \in \Sigma^+$, $q \in F$ and $s_i \in F_i$ such that $xy_1^* z_1 \cdots y_m^* z_m u^\omega \subseteq L_\omega(\mathcal{M}, \{F\})$, $F_i = Q(s_i, y_i)$ for each $1 \leq i \leq m$, (s_i, y_i) is a simple cycle of $\mathcal{M}$,

$$q_0 \cdot x = s_1,\ s_1 \cdot z_1 = s_2, \ldots,\ s_{m-1} \cdot z_{m-1} = s_m,\ s_m \cdot z_m = q,\ Q(q, u) = F,$$

and $y_i(0) \neq z_i(0)$ for each $1 \leq i \leq m$. Clearly, L is the union of all such languages $xy_1^* z_1 \cdots y_m^* z_m u^\omega$. Since there are only finitely many such chains, L has the desired representation.

3→1. Follows from Lemma 12. □

Corollary 16. *The class of regular ω-languages $F_\omega(\mathbb{A}_\Sigma)$ is decidable.*

The last theorem and the corresponding fact in [10] imply the following:

Corollary 17. *The class $F_\omega(\mathbb{A}_\Sigma)$ coincides with the class of finite unions of ω-languages $V \cdot u^\omega$ where V is a sparse aperiodic language over Σ and $u \in \Sigma^+$ is a primitive word.*

Comparing the last theorem with the invariants of the Wagner hierarchy in [18] we immediately obtain the following

Corollary 18. *The ω-languages in $F_\omega(\mathbb{A}_\Sigma)$ occupy precisely the finite levels of the Wagner hierarchy.*

Next we characterize $F_k(\mathbb{A}_{\Sigma,\omega})$ for any $k < \omega$. Note that $F_0(\mathbb{A}_{\Sigma,\omega}) = \{\emptyset\}$ and $F_1(\mathbb{A}_{\Sigma,\omega})$ is the class of finite regular ω-languages over Σ.

Theorem 19. *For any $k < \omega$ and $L \in \mathcal{A}_{\Sigma,\omega}$ the following conditions are equivalent:*

1. *$L \in F_{k+2}(\mathbb{A}_{\Sigma,\omega})$.*
2. *L is recognized by a counter-free Muller acceptor that has neither ω-pattern nor a chain $F_0 < \cdots < F_{k+1} < F \in \mathcal{F}$ of Büchi macrostates.*
3. *L is a finite union of sets $xy_0^* z_0 \cdots y_k^* z_k u^\omega$ where $x, y_i, z_i \in \Sigma^*$, $u \in \Sigma^+$ and $y_0^*, \cdots, y_k^* \in \mathcal{A}_\Sigma$.*

Proof. 1→2. Let $(\mathcal{M}, \mathcal{F})$ be a counter-free Muller acceptor recognizing L. By Lemma 14, $(\mathcal{M}, \mathcal{F})$ has no ω-pattern. Suppose for a contradiction that $(\mathcal{M}, \mathcal{F})$ has a chain $F_0 < \cdots < F_{k+1} < F \in \mathcal{F}$ of Büchi macrostates. In notation of the previous proof, $xy_0^* z_0 \cdots y_{k+1}^* z_{k+1} u^\omega \subseteq L$. Since $y_i(0) \neq z_i(0)$ for each $i \leq k+1$,

$xy_0^*z_0\cdots y_{k+1}^*z_{k+1}u^\omega \not\in F_{k+2}(\mathbb{A}_{\Sigma,\omega})$ by Lemma 13, hence also $L \not\in F_{k+2}(\mathbb{A}_{\Sigma,\omega})$. A contradiction.

2→3. The desired representation of L follows from the proof of implication 2→3 in the previous theorem.

3→1. Follows from Lemma 12. □

Corollary 20. *For any $k < \omega$, the class of regular ω-languages $F_k(\mathbb{A}_{\Sigma,\omega})$ is decidable.*

Now we are able to prove the item 2 of the main theorem in Introduction.

Theorem 21. *For any alphabet Σ with at least two symbols we have:*

$$F_0(\mathbb{A}_{\Sigma,\omega}) \subset F_1(\mathbb{A}_{\Sigma,\omega}) \subset \cdots \subset F_\omega(\mathbb{A}_{\Sigma,\omega}) = F_{\omega+1}(\mathbb{A}_{\Sigma,\omega}),$$

for each $n < \omega$ the BA $\mathbb{A}_{\Sigma,\omega}^{(n)}$ is atomic with infinitely many atoms, and $\mathbb{A}_{\Sigma,\omega}^{(\omega)}$ is a countable atomless BA.

Proof. First we check that $F_k(\mathbb{A}_{\Sigma,\omega}) \subset F_{k+1}(\mathbb{A}_{\Sigma,\omega})$ for each $k < \omega$. For $k = 0$ the inclusion is trivial. Let $a, b \in \Sigma$, $a \neq b$. It suffices to show that $a^*ba^\omega \in F_2(\mathbb{A}_\Sigma) \setminus F_1(\mathbb{A}_\Sigma)$,

$$a^*ba^*ba^\omega \in F_3(\mathbb{A}_\Sigma) \setminus F_2(\mathbb{A}_\Sigma),\ a^*ba^*ba^*ba^\omega \in F_4(\mathbb{A}_\Sigma) \setminus F_3(\mathbb{A}_\Sigma),$$

and so on. By Theorem 15 and Lemma 12,

$$a^*ba^\omega \in F_2(\mathbb{A}_\Sigma),\ a^*ba^*ba^\omega \in F_3(\mathbb{A}_\Sigma),\ a^*ba^*ba^*ba^\omega \in F_4(\mathbb{A}_\Sigma),$$

and so on. By Lemma 13,

$$a^*ba^\omega \not\in F_1(\mathbb{A}_\Sigma),\ a^*ba^*ba^\omega \not\in F_2(\mathbb{A}_\Sigma),\ a^*ba^*ba^*ba^\omega \not\in F_3(\mathbb{A}_\Sigma),$$

and so on.

By Lemmas 12 and 13, elements

$$a^*ba^\omega/F_1(\mathbb{A}_{\Sigma,\omega}), a^*ba^*ba^\omega/F_2(\mathbb{A}_{\Sigma,\omega}), \ldots$$

are atoms respectively in $\mathbb{A}_{\Sigma,\omega}^{(1)}, \mathbb{A}_{\Sigma,\omega}^{(2)}, \ldots$, and, for each $n < \omega$, the same applies to the elements

$$b^na^*ba^\omega/F_1(\mathbb{A}_{\Sigma,\omega}), b^na^*ba^*ba^\omega/F_2(\mathbb{A}_{\Sigma,\omega}), \ldots.$$

Since the languages $b^na^*ba^\omega$ (as well as the languages $b^na^*ba^*ba^\omega$ and so on) are pairwise disjoint for distinct n, the BA's $\mathbb{A}_{\Sigma,\omega}^{(1)}, \mathbb{A}_{\Sigma,\omega}^{(2)}, \ldots$ have infinitely many atoms (as well as the BA $\mathbb{A}_{\Sigma,\omega}^{(0)} = \mathbb{A}_{\Sigma,\omega}$).

Next we check that the BA $\mathbb{A}_{\Sigma,\omega}^{(k)}$ is atomic for each $k < \omega$. For $k < 2$ this is again obvious, so it remains to show that for any $k < \omega$ and $L \in \mathcal{A}_{\Sigma,\omega} \setminus F_{k+2}(\mathbb{A}_{\Sigma,\omega})$ there is a regular aperiodic ω-language $M \subseteq L$ such that

$M/F_{k+2}(\mathbb{A}_{\Sigma,\omega})$ is an atom of $\mathbb{A}^{(k+2)}_{\Sigma,\omega}$. We distinguish the cases $L \not\in F_\omega(\mathbb{A}_{\Sigma,\omega})$ and $L \in F_\omega(\mathbb{A}_{\Sigma,\omega})$. Let $(\mathcal{M}, \mathcal{F})$ be a counter-free Muller acceptor recognizing L.

In the first case, by Theorem 15 $(\mathcal{M}, \mathcal{F})$ has an ω-pattern, hence M exists by the proofs of Lemmas 14 and 13. In the second case, by Theorem 19 there is a chain $F_0 < \cdots < F_{k+1} < F \in \mathcal{F}$ of Büchi macrostates and the words specified there, so in particular $z_i(0) \neq y_i(0)$ for each $i \leq k+1$. Then the ω-language $M = xy_0^*z_0 \cdots y_{k+1}^*z_{k+1}u^\omega$ has the desired property.

It remains to show that for any $L \in \mathcal{A}_{\Sigma,\omega} \setminus F_\omega(\mathbb{A}_{\Sigma,\omega})$ there is a regular aperiodic ω-language $M \subseteq L$ such that $M, L \setminus M \not\in F_\omega(\mathbb{A}_{\Sigma,\omega})$. Let again $(\mathcal{M}, \mathcal{F})$ be a counter-free Muller acceptor recognizing L. By Theorem 15, $(\mathcal{M}, \mathcal{F})$ has an ω-pattern with the corresponding words x, v_1, v_2, w, u (see text before Lemma 14). Since $(\mathcal{M}, \mathcal{F})$ is counter-free, $v_1^*, v_2^* \in \mathcal{M}_\Sigma$. By the proof of Lemma 14, we can take $M = xv_1(v_1+v_2)^*wu^\omega$. □

Similarly to Corollary 10 we obtain

Corollary 22. *For any alphabet Σ, $\mathbb{A}_{\Sigma,\leq\omega} \simeq \mathbb{A}_\Sigma$.*

References

1. Choffrut, C., Karhumäki, J.: Combinatorics of words. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 329–438. Springer (1997)
2. Gehrke, M., Grigorieff, S., Pin, J.-É.: Duality and Equational Theory of Regular Languages. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 246–257. Springer, Heidelberg (2008)
3. Goncharov, S.S.: Countable Boolean Algebras and Decidability. Plenum, New York (1996)
4. Hanf, W.: The boolean algebra of logic. Bull. Amer. Math. Soc. 20(4), 456–502 (1975)
5. Lempp, S., Peretyat'kin, M., Solomon, R.: The lindenbaum algebra of the theory of the class of all finite models. Journal of Mathematical Logic 2(2), 145–225 (2002)
6. Marini, C., Sorbi, A., Simi, G., Sorrentino, M.: A note on algebras of languages. Theoretical Computer Science 412, 6531–6536 (2011)
7. Perrin, D., Pin, J.E.: Infinite Words. Applied Mathematics, vol. 141. Elsevier, Amsterdam (2004)
8. Pin, J.E.: On regular languages (2011) (unpublished manuscript)
9. Pippenger, N.: Regular languages and stone duality. Theory of Computing Systems 30(2), 121–134 (1997)
10. Selivanov, V., Konovalov, A.: Boolean Algebras of Regular Languages. In: Mauri, G., Leporati, A. (eds.) DLT 2011. LNCS, vol. 6795, pp. 386–396. Springer, Heidelberg (2011)
11. Selivanov, V.L.: Hierarchies, numerations, index sets (1992) (unpublished manuscript, 290 p.)
12. Selivanov, V.L.: Universal boolean algebras with applications. In: Abstracts of Int. Conf. in Algebras, p. 127. Institute of Mathematicsk, Novosibirsk (1997) (in Russian)
13. Selivanov, V.L.: Positive structures. In: Cooper, S., Goncharov, S. (eds.) Computability and Models, Perspectives East and West, pp. 321–350. Plenum (2003)

14. Sikorski, R.: Boolean Algebras. Springer, Berlin (1964)
15. Straubing, H.: Finite automata, formal logic and circuit complexity. Birkhäuser, Boston (1994)
16. Szilard, A., Yu, S., Zhang, K., Shallit, J.: Characterizing Regular Languages with Polynomial Densities. In: Havel, I.M., Koubek, V. (eds.) MFCS 1992. LNCS, vol. 629, pp. 494–503. Springer, Heidelberg (1992)
17. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 389–455. Springer (1996)
18. Wagner, K.: On ω-regular sets. Information and Control 43, 123–177 (1979)
19. Yu, S.: Regular languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 41–110. Springer (1997)

Pumping, Shrinking and Pronouns: From Context Free to Indexed Grammars

Eli Shamir

School of Computer Science and Engineering
The Hebrew University of Jerusalem, Jerusalem 91904, Israel
shamir@cs.huji.ac.il

Abstract. Pumping-shrinking operation is a useful tool in the study of Formal Languages. A pump-shrink bound for Linear Indexed Grammars is derived, depending only on the the number of auxiliary symbols of the grammar. Observations are suggested on formal operations analogous to shrinking of subordinate clauses.

Keywords: Formal Languages, pumping shrinking lemmas, Linear-Index Grammars.

1 Introduction

The pumping "$xuwvy$ Lemma" for Context-Free Language (CFL) turned out to be the most popular result among many results about the CFL model proved in the seminal 1961 article [2].

Several variants improving the Lemma were proved [9]. The Lemma gives a useful tool to exhibit the limitations of the CFL model, showing it cannot tackle cross copying and triple counts of symbols. It also serves to prove a variety of results about inherent ambiguity of certain CFLs [11].

Actually, the shrinking of $xuwvy$ to xwy suffices for several applications and linguistically, makes more sense: long and complicated sentences contain subordinate clauses which can be shrinked, thus revealing the skeleton structure of the main sentence. However, in natural languages, a subordinate clause (say of noun-phrase category) does not shrink to the empty string but to a *pronoun* or a simple noun. In a similar vein, we call *pro-shrinking* an operation on derivation trees which replaces a subtree with root labeled A by a smaller (or smallest) tree with root A and generating a terminal string. A collection of such trees (for various A's) is obtained by running the bottom-up emptiness-checking algorithm for G (see Section 2). Such a collection is a formal analogue of treebanks, compiled in order to facilitate computer-aided parsing, even under mild context-sensitivity [8].

A "mildly context sensitive" grammar model is required to satisfy certain formal properties [16], some of which are related to, and proved by, pumping or (pro-)shrinking. A particularly pleasant family of formal languages [15] has four weakly equivalent grammar-models, including the Linear Index Grammar (LIG) model.

Section 3 is the technical section. We define the Indexed Grammar model, but concentrate on the Linear IG model. We present proofs and bounds of the pumping and shrinking in LIG, which are quite simpler than the existing proofs for the alternative weakly equivalent "tree-adjoining" model.

A derivation in LIG has a context-free-like tree. Each node of the tree is labeled by an index which behaves like a stack (pushdown) as one proceeds along branches (or "fibers") of the tree. In the linear (LIG) model the stack is carried from a father node to just a single child node. This renders the stacks along the fibers independent, unrelated, and facilitates the proof of the pumping and shrinking results.

In section 4 we discuss the more powerful (non-linear) Index Grammar (IG) [1].

We comment on the complex statements and proofs of pumping and shrinking Lemmas which were given. We argue pro-shrinking may be easier to apply if one puts an Indexed grammar or a sub-model of it to practical use.

2 Context Free Pumping, Shrinking and Pro-Shrinking

We assume throughout a binary normal form for the productions (see also note below).

$$\begin{array}{ll} A \to BC & \text{for non-terminal productions,} \\ A \to a & \text{for terminating productions.} \end{array} \tag{1}$$

The pumping bound *pumpsh*(G) is the size of terminal strings *beyond which* pumping and shrinking must appear. Indeed for n-state automaton, even non-deterministic, this bound is $\leq n$. For context-free G with n non-terminal symbols, pumping and shrinking relies on the repetition of a non-terminal label A on one branch of the derivation tree. So if the height of a tree generating z in $> n = |V(N)|$, the number of non-terminals in G, then the z admits a CFG-type pump-shrink, i.e., $z = xuwvy$ and

$$z(k) = xu^k wv^k y \in L(G) \text{ for } k = 0 \ (shrink), 1, 2, \ldots. \tag{2}$$

Since the full binary tree is possible (but rare in practice [10]), the bound for *pumpsh*(G) is 2^n. Pro-shrinking is clearly more drastic; presumably it can drive $sh(G)$-the size of the smallest string in $L(G)$ - below 2^n, but no general proof for all grammars is known.

A collection of small trees $\{\text{pro}(A), A \text{ in } V(N)\}$, is traced while running the bottom-up "algorithm for emptiness" of $L(G)$:

$$N(0) = T; \;\; N(i+1) = N(i) \cup \{\text{all } A \text{ where } \; A \to BC \text{ and } B, C \in \text{N(i)}\}; \tag{3}$$

termination occurs when $N(i)$ stops growing, after $\leq n$ steps, where $n = |V(N)|$ the number of non-terminals in G. At the first time A enters $N(i)$, its smallest subtree - and terminal string it derives - is actually traced.

In grammar models extending CF, representation of derivation by labeled trees is usually preserved, but the set of labels for the nodes is potentially unbounded. Hence the termination of the emptiness algorithm, bounding the set of labels used in a run, shrinking and pro-shrinking issues - are all intertwined, as we shall see in detail for the Linear Index Grammar models in the next section.

Note: We add here a didactic note about isomorphism of pushdown machine walks and leftmost generation in context free grammars.

Consider the following S-normal form (S for stack) of CF grammars which unifies several known normal forms, and is easy to obtain. All productions of G are of the form:

(i) $A \to a$
(ii) $A \to aBC$
(iii) $A \to BC$

Capitals denote non-terminal symbols, lower case letters are terminals. At each step of a leftmost derivation, the leftmost non-terminal is rewritten according to one of the productions, until a string of terminal is obtained.

This corresponds to a *step by step* walk of a pushdown (stack) automation, with capital letters as stack symbols. Namely:

(i) Upon reading the input symbol a, the top symbol A is *popped.*
(ii-iii) Upon reading the input symbol a (or the empty one is case of (iii)) the stack symbols BC are *pushed* to the top instead of A.

The accepting walk of the pushdown automation starts with a stack containing S alone and concludes when the stack is empty.

3 Pumping-Shrinking for Linear Indexed Grammars

The model of indexed grammars [IG] introduced by [1], replaces CF productions by production schemes (4,5). In IG, the non-terminal symbols $A, B, C, \ldots \in V(N)$ carry an index which the productions treat and change in a stack (push-down) manner, with stack symbols $f, g, h, \dots \in V(I) \cup \{\varepsilon\}$ ($\varepsilon =$ the empty string). Without loss of generality, production schemes are in binary normal form:

$$A(f\gamma] \to B[gh\gamma]C[gh\gamma]; \tag{4}$$

$$A[\ ] \to a \ \{\text{unary production to terminals } a, b, c, \dots \in V(T)\}, \tag{5}$$

the brackets in (4) contains the stack, with top symbol on the left, $\gamma, \delta, \dots \in V(I)^*$ denote strings of stack symbols, $[\quad]$ denotes an empty stack. If $g = h = \varepsilon$, $\mathbf{f} \neq \varepsilon$ then (4) is a "pop stack" production. If f, g, h all $\neq \varepsilon$, (4) "pushes" gh on top of stack instead of f, increasing stack-depth by 1.

Note that (A, f) is the enabling "state" for the scheme (4) to apply, for any γ. The notion of a derivation (or generation) in Indexed Grammars in the same as in CF: A sequence of sentential forms, starting from $S[\ ]$ to a terminal string on $V(T)^*$. The same holds for the derivation tree, with nodes labeled by indexed non-terminals (a potentially unbounded set).

Note 1. It suffices to consider the binary tree from the root $S[\ ]$ to "pre-terminal" leaves $A[\ ]$ which produce terminals only via (5).

The IG model has a considerable generative power, but difficult to apply and study formally (see Section 4). The more restrictive model of Linear-Indexed-Grammars (LIG), introduced in [3] turned out to be linguistically applicable, to satisfy the features required from "mildly context-sensitive" model [16], and to be weakly equivalent (in generation power) to three alternative grammar models [15].

Instead of (4), the production schemes in LIG are

$$A[f,\gamma] \to B[gh\gamma]C[\ ], \quad \textbf{or} \qquad \to B[\ ]C[gh\gamma]; \tag{6}$$

i.e., the index passes (with due change) *to only one child*, the *blue*(blood) *child*. The other, *red child*, gets an empty stack (which subsequently can grow in the descendants). Linearity *eliminates the rigid dependence* between stacks evolutions on distinct branches of the tree. This renders the linguistic application and formal study of LIG much simpler: Polynomial time parsing algorithms [14]; and shrink-pump (P-S) properties as we *present in detail here* - previous studies used the weakly equivalent tree adjoining model [13].

Pump-shrink comes from certain repetitions on branches of the binary derivation tree.

Theorem 1 (The pump-shrink (P-S) Theorem for LIG). *Let G be a LIG grammar. If $z \in L(G)$ is longer than the constant $Pumpsh(G)$ (which is bounded in (8) below), then z admits a pump-shrink which is either of CF-type (see (2)) or of trapezoid type (see (12) below).*

Proof. Notice that if there is a suitable bound on blue segments in branches of the derivation tree, then Lemma 1 provides a bound which assures a CF-type P-S along a branch with enough red nodes.

Lemma 1. *In a labeled binary tree where one child is blue and the other is red, let Γ = max length of blue segment in a branch. If*

$$\|T\| = the\ total\ number\ of\ nodes\ on\ the\ tree\ > 2^r\Gamma^r \tag{7}$$

then there is a branch with $> r$ red nodes.

Proof. We look for a branch $(B^*R)^{r+1}$. The tree is peeled in stages:

Stage 0. Take the (longest) blue segment from the root.
Stage 2i-1. Take red children on nodes in stage 2i-2.
Stage 2i. Take red children and blue branch-segments below the red nodes of stage 2i-1.

This peeling terminates when all nodes in a stage are leaves. Upon termination, all nodes were peeled off (easy induction). Stages 2i-1 and 2i together multiply n number of nodes by $2 \cdot \Gamma$ at most. After $2r$ stages the bound is $(2\Gamma)^r$. Once we enter the $2r + 1$ stage, a branch with $(r + 1)$ reds appears.

It remains to deal with too long blue branch. Lemma 2 (below) shows that stack-depth along a blue branch which exceeds a constant $|V|$, defined in (9), implies a trapezoid P-S. This depth bound implies a bound (exponential in $|V|$) on the length of the branch, a bound which counts the number Δ of all full labels with stack bounded by $|V|$. Indeed in branches longer than Δ the same full label will occur twice, leading again to a CF-type pump-shrink.

Combining the bounds from the two Lemmas, we get the bound

$$Pumpsh(G) \leq 2|V(N)|^{|V(N)|}|V(I)|^{|V(N)|^3|V(I)|^2}. \tag{8}$$

To conclude the proof we state and prove Lemma 2.

Lemma 2. *If the max stack depth along an (all) blue branch (starting and ending in empty stack) exceeds*

$$|V| = (|V(N)||V(I)|)^2 = \{\text{total number of pairs of enabling states in (6)}\}, \tag{9}$$

then this branch contains a "trapezoid" P-S configuration (explicated in the proof below).

Proof of Lemma 2. Let $t = 0, 1, 2, \cdots$ parametrize the sequence of nodes along the blue branch segment. A *trapezoid configuration* at $t_1 < t_2 < t_3 < t_4$ is a quadruple of full labels

$$\begin{array}{ll} t_1 : C[f\gamma] & \qquad\qquad t_4 : D(g\gamma) \\ \\ \quad t_2 : C[f\delta\gamma] & t_3 : D[g\delta\gamma] \end{array} \tag{10}$$

$$Moreover\ the\ stack\ walk\ from\ t_1\ to\ t_2, and\ from\ t_3\ to\ t_4\ does\ not\ touch\ the$$

$$stack\ bottom\ \gamma;\ the\ walk\ from\ t_2\ to\ t_3\ does\ not\ touch\ the\ bottom\ \delta\gamma. \tag{11}$$

The condition (11) is clearly met if the stack depth behaves in a unimodal manner, increasing to a maximum and then decreasing. In general (11) will hold if we choose $t_1, \cdots, t_4$ from a subsequence of t-values defined as follows: from the max-depth value H, move t up and down to the closest values where the depth is $H - 1$, then to closest values it assumes $H - 2$, and so on.

Now if the depth gap in stack walk exceeds $|V|$ in (9), simple count shows that a trapezoid configuration (10) exists, i.e. enabling states (C, f) upward paired with (D, g) downword repeat at two different depth values (horizontal cuts) of the depth graph. Now the shrink is obtained by dropping the up-depth segment $t_1, \ldots, t_2 - 1$ and the down-depth segment $t_3, \ldots, t_4 - 1$, and dropping the side-subtrees emanating from them via the red nodes on the left or right sides. It is readily verified that the shrinked derivative (tree) is legal, and winds up in a shrinked terminal string $z(0) = xw_1ww_2y$, where $z = z(1) = xuw_1u'wvw_2v'y$ is the original string and

$$z(k) = xu^kw_1u'^kwv^kw_2v'^ky \quad k = 0, 1, 2, \cdots \tag{12}$$

For k-pump, instead of once, we loop k times:

$$(t_1, \ldots, t_2 - 1)^k \text{ and } (t_3, \ldots, t_4 - 1)^k \tag{13}$$

along with all the adjoining side branches. Again this is a perfectly legal derivation. □

As in the CFG case, (12) is used to prove the limitations of the LIG-model.

The derivation tree for LIG abounds with red nodes, with empty stack. For any two of those with the same non-terminal, say B[ ], their subtrees can replace each other, or be replaced by a standard minimal tree (proB[ ]). Hence it is very likely that sh(G) the shortest terminal string in $L(G)$ is much smaller than the bound we got for *pumpsh*(G), but it is not clear how to get a better estimate valid for all LIGs.

The bottom up emptiness algorithm for LIG is essentially the same as for CFG (see (3)). It traces the minimal trees and terminal strings not only for A[ ], $A \in V(N)$, but also for $A[\gamma]$. Termination of the Algorithm (but with exponential time-bound) is assured by the $|V|$ bound (9) on the stack depth.

4 Indexed Grammar (IG) and Pro-Shrinking

Shrinking for the IG model was dealt in [5] and [4]. The later one proves a strong and elegant shrinking statement which implies linear growth of sizes in $L(G)$ and also the corresponding result for the Parikh map which counts terminal symbols separately. But it is an existence proof for a particular G which uses the axiom of choice and does not give any bound for the required size of terminal string in terms of the parameters of G, as we did in section 3 for the linear IG model.

The earlier result [5] seems to rely on the algorithm in [1] for non-emptiness and membership in $L[G]$. No estimates are given for the shrinking bound or the pumping bound in terms of the parameters of the grammar.

We conjecture that in restricted models of IG, like the one presented in [7,12], pump (G) and sh(G) will be bounded by the parameters of the grammar *provided* these models admit an efficient emptiness and membership algorithm, efficient in terms of the parameters of G.

It seems that strict shrinking (like [4],[5]) will not do it, but *pro-shrinking process* is needed, in which a big subtree of long derivation will be replaced by smaller subtree *which also lead to terminals* (this is the main difficulty).

Remark. In a grammar of IG type is put to practical use as CFG or LIG, one can compile a tree-bank of “pronouns” for subtrees with roots labeled A[Index], similar to tree-banks corpora compiled to facilitate the construction to parse trees for the task of natural language syntactic analysis.

References

1. Aho, A.V.: Indexed grammars. J. Assoc. Comput. Mach. 15, 647–671 (1968)
2. Bar-Hillel, Y., Perles, M., Shamir, E.: On formal properties of simple Phrase Structure Grammars. Z. Phonetik Sprachwiss. Kommunikat. 14, 143–172 (1961)
3. Gazdar, G.: Applicability of Indexed Grammars to Natural Languages. In: Reyle, U., Rohrer, C. (eds.) Natural Language Parsing and Linguistic Theories, pp. 69–94 (1988)
4. Gilman, R.: A Shrinking Lemma for Indexed Languages. Theo. Comp. Sci. 16, 277–281 (1996)
5. Hayashi, T.: On Derivation Trees of Indexed Grammars: An extension of the wvwxy-Theorem. Publ. RIMS Kyoto Univ. 9, 61–92 (1973)
6. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
7. Keller, B., Weir, D.: A Tractable Extension of Linear Indexed Grammars. In: Proc. EACL, pp. 75–82 (1995)
8. Maier, W., Sogaard, A.: Treebanks and Mild Context - Sensitivity. In: Proc. of Conference on Formal Grammar (FG 2008), pp. 16–76 (2008)
9. Ogden, W.: Intercalation theorems for Stack Languages. In: Proc. First Annual ACM Symposium of the Theory of Computing, pp. 31–42 (1969)
10. Saginer, Y.: Fast unsupervised Incremental Parsing. In: Proc. ACL, vol. 45, pp. 384–389 (2007)
11. Shamir, E.: Some Inherently ambiguous Context-Free Languages. Inf. and Control 18, 355–363 (1971)
12. Staudacher, P.: New frontiers beyond Context-Freeness: DI-grammars and DI-automata. In: Proc. 6th EACL (1993)
13. Vijay-Shanker, K.: A study of Tree Adjoining Grammars. Ph.D. Thesis, U. Penn (1988)
14. Vijay-Shanker, K., Weir, D.J.: Parsing some constrained grammar-formalisms. Computational Linguistics 19, 591–636 (1993)
15. Vijay-Shanker, K., Weir, D.J.: The equivalence of four extensions of Context-Free Grammars. Math. Sys. Theory 27, 511–546 (1994)
16. Vijay-Shanker, K., Weir, D.J., Joshi, A.K.: Characterizing structural descriptions produced by various grammatical formalisms. In: Proc. 25th ACL, pp. 104–111 (1987)

Online Matching of Multiple Regular Patterns with Gaps and Character Classes*

Seppo Sippu and Eljas Soisalon-Soininen

Department of Computer Science and Engineering, Aalto University,
P.O. Box 15400, FI-00076 Aalto, Finland
{seppo.sippu,eljas.soisalon-soininen}@aalto.fi

Abstract. Given a dictionary D of regular expressions and a text T, the online regular-pattern-matching problem is to single out, for each text position $T[c]$, those expressions in D that have a match ending at $T[c]$, while processing T only once. This problem is considered in the context of regular patterns over bounded-length gaps and keywords, where the gaps are specified by wildcards and character classes and the keywords are strings over the input alphabet. Our algorithm is based on constructing the Aho–Corasick pattern-matching automaton for the set of keywords, and representing as a bit vector the set of keywords that can precede a given keyword in a regular-pattern instance. For a dictionary D with r patterns and with k_i keywords in pattern i, the preprocessing takes time $O(|D| + \sum_{i=1}^{r} k_i^2 \log k_i/w)$, where w denotes the number of bits in a memory word. When only fixed-length wildcard gaps without character classes are allowed, the time spent by our matching algorithm for each text character $T[c]$ is at most $O((\log r + k/w)(K_c + 1))$, where $k = \max\{k_1, \ldots, k_r\}$ and K_c is the number of keyword occurrences in D matched at text position $T[c]$.

Keywords: string processing algorithms, online dictionary matching, regular pattern matching, variable-length gaps, character classes.

1 Introduction

Regular-expression matching means matching of the strings defined by the expression. In other words, given a regular expression R and a text T we want to find those strings S that are substrings of T and belong to the language $L(R)$. This problem can be alternatively stated as finding those prefixes of T that are defined by the expression $\Sigma^* R$, where Σ^* is the reflexive transitive closure of the alphabet Σ. Thus, in order to find efficient algorithms for regular expression matching it is necessary (and sufficient) to devise algorithms for efficient recognition of regular languages.

Efficiency is required not only in the length n of text T, but also in the size m of R. Thus it is not feasible to construct a deterministic finite automaton from R, and to feed T to this automaton, because this solution—though efficient in n—is

* This work was supported by the Academy of Finland.

exponential in m. This is especially true in the case in which there is a large dictionary of regular patterns to be matched, and an online solution is required for multi-pattern matching, so that all the patterns in the dictionary are matched in a single scan of text T. Such a scenario is typical in many applications such as XML quering and filtering, and in internet traffic analysis.

A nondeterministic finite automaton can be constructed from regular expression R of size m in time $O(m)$ using the standard textbook solution originally presented by Thompson [11]. Using the nondeterministic automaton the language-recognition problem can be solved in time $O(mn)$ and space $O(m)$ with preprocessing (automaton construction) in time $O(m)$ only.

Several improvements upon the bounds by Thompson [11] have been presented [2, 3, 5, 6, 8]. In a recent article by Bille and Thorup [6] the time bound obtained is

$$O(n\frac{k\log w}{w} + m), \tag{1}$$

where k is the number of *keywords*, that is, maximal substrings of the pattern that contain only input characters, and w is the number of bits in a memory word. (In this paper we assume a fixed-sized alphabet, and we thus simplified the bound by Bille and Thorup [6] accordingly.)

The contribution of Bille and Thorup [6] is two-fold. First, using multi-string matching of Aho–Corasick [1] together with the simulation of the nondeterministic automaton with keyword numbers instead of characters, they managed to get the bound $O(nk + m)$. Second, they applied to this algorithm the strategy of Bille [2] to decompose the nondeterministic automaton into micro automata each of which contains only w states, thus yielding the bound (1).

In the present paper we devise a regular-expression matching algorithm that directly uses the Aho–Corasick multi-string matching automaton constructed from the keywords. The idea is to collect matches of prefixes of instances of regular patterns found in the text. Whenever a new keyword occurrence is recognized the algorithm checks whether or not this occurrence forms a legal continuation of an instance prefix found thus far; and if so, inserts the keyword with the information of the position into the collection. To make this checking possible we precompute for each keyword in a pattern the set of keywords that may precede that keyword in an instance of the pattern; each such set is stored as a bit vector of k bits, for a pattern with k keywords.

In the case of one pattern only, after preprocessing in time $O(m + k^2 \log k/w)$, our algorithm spends for each text character $T[c]$ at most time

$$O(\frac{k}{w}(K_c + 1)),$$

where K_c is the number of keywords matched at text position $T[c]$. An upper bound for K_c is the maximal number of suffixes of a single keyword that are also keywords.

The algorithm of Bille and Thorup [6] spends time $O(k \log w/w)$ per character, and thus our algorithm is better if $K_c < \log w$ (assuming that there is no significant difference in the hidden constants in the complexity bounds). For example, consider the following regular expression used to detect a Gnutella data download signature in a stream [10]:

```
(Server:|User-Agent:)( |\t)*(LimeWire|
BearShare|Gnucleus|Morpheus|XoloX|
gtk-gnutella|Mutella|MyNapster|Qtella|
AquaLime|NapShare|Comback|PHEX|SwapNut|
FreeWire|Openext|Toadnode)
```

In this expression no keyword is a suffix of another, and thus $K_c \leq 1$ at all text positions $T[c]$, but for $w = 64$, $\log w = 6$.

However, the main advantage of our algorithm is that it solves the more general problem of *online dictionary matching* of regular patterns, where, given a dictionary D of patterns and a text T, the text T is scanned only once and, at each text position $T[c]$, exactly the patterns in D that have a match ending at $T[c]$ are reported. Moreover, besides keywords composed of characters in the input alphabet we allow the patterns to contain bounded-length *gaps* specified by wildcards and character classes. The gaps may be of varying length, but any character classes contained in a gap must be in fixed positions in the instances of the gap.

For a dictionary D with r patterns and with k_i keywords in pattern i, $i = 1, \ldots, r$, the preprocessing phase of our algorithm takes time

$$O(|D| + \sum_{i=1}^{r} k_i^2 \log k_i / w).$$

When only fixed-length wildcard gaps without character classes are allowed, the time spent by our matching algorithm for each text character $T[c]$ is at most

$$O(\log r + \frac{k}{w})(K_c + 1)),$$

where $k = \max\{k_1, \ldots, k_r\}$ and K_c is the number of keywords (the same keyword string counted as many times it occurs in D) matched at text position $T[c]$. When wildcard gaps of variable length and character-class gaps of fixed-length are allowed, then in the above bound the factor $\log r + k/w$ must be replaced by $(d_1 + 1)(\log r + k/w) + d_2$, where d_1 is the maximal difference of the maximum and minimum lengths of a gap and d_2 is the maximal number of character classes in a gap.

2 Regular Patterns with Gaps

Assume that we are given a string T of length $|T| = n$ (called the *text*) over a character alphabet Σ, whose size is assumed to be fixed, and a finite set D (called the *dictionary*) of *patterns*. Each pattern P_i is a regular expression over strings gw, where g is a *wildcard gap* of the form "$.\{l, h\}$" and w is a *keyword* in Σ^*. For integers $h \geq l \geq 0$, the gap ".$\{l, h\}$" matches any string of length l to h in Σ^*. The gap ".$\{0, 0\}$" can also be denoted by the empty string (ϵ), the gap ".$\{1, 1\}$ by ".", the gap ".$\{2, 2\}$ by "..", etc. The regular operators allowed in the patterns are concatenation, union ($|$), and iteration (*), and parentheses can be used to enclose subexpressions, as usual.

For example, the pattern $..ab.\{1, 3\}c.d$ is of the form $g_1w_1g_2w_2g_3w_3$, where the gaps are $g_1 = ..$, $g_2 = .\{1, 3\}$, $g_3 = .$, and the keywords are $w_1 = ab$, $w_2 = c$, $w_3 = d$. This pattern matches with, say, the input text *eeeabeecedeee*. The pattern $aa(bb|.\{1, 3\}c)^*d = g_1w_1(g_2w_2|g_3w_3)^*g_4w_4$ has the same gaps and keywords as the pattern $.\{0, 0\}aa(.\{0, 0\}bb|.\{1, 3\}c)^*.\{0, 0\}d$, namely the gaps $g_1 = .\{0, 0\}$, $g_2 = .\{0, 0\}$, $g_3 = .\{1, 3\}$, $g_4 = .\{0, 0\}$, and the keywords $w_1 = aa$, $w_2 = bb$, $w_3 = c$, $w_4 = d$. The pattern $a.^*c = g_1w_1(g_2w_2)^*g_3w_3$ has the gaps $g_1 = \epsilon$, $g_2 = .$, $g_3 = \epsilon$, and the keywords $w_1 = a$, $w_2 = \epsilon$, $w_3 = c$.

Our task is to determine all occurrences of all patterns $P_i \in D$ in text T. We report a pattern occurrence by a pair of a pattern number and the character position in T of the last character of the occurrence. A pattern may have many occurrences that end at the same character position; all these occurrences are reported once by the same pair of pattern number and character position.

For each pattern P_i, we number the occurrences of its gaps and keywords, so that $gap(i, j)$ denotes the jth gap and $keyword(i, j)$ denotes the jth keyword, that is, the keyword following $gap(i, j)$, $i = 1, 2, \ldots, k_i$. Strings that appear as the jth keyword for some j are called *keyword strings*. This distinction is necessary because the same keyword string can appear in many positions, and thus the number of keyword strings in a pattern can be less than k_i.

For pattern P_i, we denote by $mingap(i, j)$ and $maxgap(i, j)$, respectively, the minimum and maximum lengths of strings in Σ^* that can be matched by $gap(i, j)$. The length of the jth keyword of pattern P_i is denoted by $length(i, j)$. For example, for the pattern $P_i = aa(bb|.\{1, 3\}c)^*d$ we have:

$$\begin{aligned}
&mingap(i, 1) = 0,\ maxgap(i, 1) = 0,\ length(i, 1) = 2,\\
&mingap(i, 2) = 0,\ maxgap(i, 2) = 0,\ length(i, 2) = 2,\\
&mingap(i, 3) = 1,\ maxgap(i, 3) = 3,\ length(i, 3) = 1,\\
&mingap(i, 4) = 0,\ maxgap(i, 4) = 0,\ length(i, 4) = 1.
\end{aligned}$$

For each pattern P_i we define the set $begins(i)$ to contain all j such that an instance of $gap(i, j)keyword(i, j)$ appears as a prefix of some instance of P_i, and the set $ends(i)$ to contain all j such that an instance of $gap(i, j)keyword(i, j)$ appears as a suffix of some instance of P_i. Furthermore, for each $j = 1, \ldots, k_i$, we define the set $precedes(i, j)$ to contain all l such that an instance of $gap(i, l)keyword(i, l)$ immediately precedes an instance of $gap(i, j)keyword(i, j)$ in some instance of P_i. For example, for the pattern $P_i = aa(bb|.\{1, 3\}c)^*d$ we have:

$$\begin{aligned}
&begins(i) = \{1\},\ ends(i) = \{4\},\\
&precedes(i, 1) = \emptyset,\ precedes(i, 2) = precedes(i, 3) = precedes(i, 4) = \{1, 2, 3\}.
\end{aligned}$$

Each of the sets $begins(i)$, $ends(i)$, and $precedes(i, j)$ can be computed from pattern P_i in linear time. If P_i contains union or iteration, the combined size of the sets may be quadratic in the number of keyword occurrences in P_i, as is the case with the pattern $(a_1|\epsilon)(a_2|\epsilon)\dots(a_k|\epsilon)$, for example. However, it is possible to construct from a general nondeterministic finite automaton (NFA) of size m an equivalent ϵ-free NFA in time $O(m \log m)$, as shown by Schnitger [9]. Thus, it is possible to construct a representation of all sets *precedes*(i, j) for pattern P_i in time $O(k_i \log k_i)$, such that these can be stored in bit vectors of k_i bits in time $O(k_i^2 \log k_i / w)$, where w is the number of bits in a memory word.

3 Multi-string Matching

For the set of all keywords in the dictionary D of regular patterns, we construct an Aho–Corasick pattern-matching automaton (PMA) [1] with an output function, represented by sets *output*(q) containing *output tuples* of the form (i, j), where $q = \mathit{state}(\mathit{keyword}(i, j))$, the state reached from the initial state upon reading the jth keyword of pattern P_i. Note that there may be several pairs (i, j) in one set *output*(q), because one keyword string may equal many different keyword occurrences in D.

The current character position, that is, the number of characters scanned from the input text is maintained in the global variable *character-count*. The operating cycle of the PMA is given as Alg. 1. The procedure *scan-next*(*character*) returns the next character from the input text. The functions *goto* and *fail* are the goto and fail functions of the standard Aho–Corasick PMA, so that $\mathit{goto}(\mathit{state}(y), a) = \mathit{state}(ya)$, where ya is a prefix of some keyword and a is in Σ, and that $\mathit{fail}(\mathit{state}(uv)) = \mathit{state}(v)$, where uv is a prefix of some keyword and v is the longest proper suffix of uv such that v is also a prefix of some keyword.

Algorithm 1. Operating cycle of the PMA.

```
state ← initial-state
character-count ← 0
traverse-output-path(state)
scan-next(character)
while character was found do
  character-count ← character-count + 1
  prefix-matches[(character-count − 1) mod maxdist + 1] ← ∅
  while goto(state, character) = fail do
    state ← fail(state)
  end while
  state ← goto(state, character)
  traverse-output-path(state)
  scan-next(character)
end while
```

The procedure call *traverse-output-path*(*state*) traverses all states q such that *string*(q), the unique string y with $\mathit{state}(y) = q$, is a keyword and a suffix of *string*(*state*). The function *output-fail*(q) that determines these states is defined by: *output-fail*(q) = *fail*$^k(q)$, where k is the greatest integer less than or equal to the length of *string*(q) such that for all $m = 1, \ldots, k-1$, the current output of *fail*$^m(q)$ is empty. Here *fail*^{m} denotes the *fail* function applied m times. Thus, the output path for state q includes those states in the fail path from q for which the output set is nonempty. The array *prefix-matches* is used to collect matches of pattern prefixes, as will be explained below.

4 Matching for Regular Patterns with Gaps

Let D be a dictionary of size m of patterns that are regular expressions over strings gw, where g is a bounded-length gap and w is a keyword, as defined above. In this section we give an algorithm that finds all occurrences of patterns in D in given text T.

As a preprocessing task we construct an Aho–Corasick pattern matching automaton from all keywords in dictionary D in time $O(m)$, and compute the sets *begins*(i), *ends*(i), and *precedes*(i, j), for all pairs *gap*(i, j)*keyword*(i, j). As explained above in Section 2, for each pattern P_i these sets can be computed and stored as bit vectors of length k_i in time $O(k_i^2 \log k_i / w)$, where k_i is the number of keywords (i, j) and w is the number of bits in a memory word.

Matches of prefixes of instances of the patterns found in the text are collected into an array *prefix-matches*, which has

$$\mathit{maxdist} = \max\{\mathit{maxgap}(i,j) + \mathit{length}(i,j) \mid i \geq 1, j \geq 1\} \qquad (2)$$

entries and stores matches found at the last *maxdist* character positions scanned from the text. The pattern-instance prefixes found at character position c are stored in the array entry

$$\mathit{prefix\text{-}matches}[(c-1) \bmod \mathit{maxdist} + 1],$$

which is initialized as empty in Alg. 1 whenever *character-count* reaches c. As the matching proceeds, the entry will contain a set of pairs (i, v), where i is a pattern number and v is a set of keyword numbers j of pattern P_i such that a match of some prefix of an instance of pattern P_i ending at keyword j has been found at position c in the text. The set of pairs (i, v) in an entry of *prefix-matches* is implemented as a balanced binary search tree indexed by i and the set v is implemented as a bit vector of k_i bits.

Given pattern number i and character position c, the function

$$\mathit{prefix\text{-}matches}(i, c)$$

searches the binary search tree at entry *prefix-matches*[$(c-1) \bmod$ *maxdist* $+ 1$] for i, and, if (i, v) is found, returns v; otherwise, the function returns the null bit vector of length k_i.

Given pattern number i, keyword number j and character position c, the procedure

$$\mathit{insert\text{-}prefix\text{-}match}(i, j, c)$$

searches the binary search tree at entry *prefix-matches*$[(c-1) \bmod \mathit{maxdist}+1]$ for i, and, if (i, v) is found, inserts j into v; otherwise, the procedure inserts $(i, \{j\})$ into the search tree.

The standard traversal of the Aho–Corasick pattern matching automaton can now be augmented to solve our matching problem for regular patterns with bounded-length gaps, as presented in Alg. 2. The **while** loop in the algorithm traverses—using the *output-fail* function as defined in Section 2—all states q at which a match of a keyword at $T[c]$ can be found. Within the **while** loop the **for** loop traverses all pairs (i, j) found in *output*(q), and checks whether or not some keyword (i, j') in the set *prefix-matches*$[(c-1) \bmod \mathit{maxdist}+1]$, where character position c is within a correct distance from *character-count*, precedes keyword (i, j).

This checking is implemented by first performing a bitwise "or" of the bit vectors returned by the function calls *prefix-matches*(i, c), $c = b, \ldots, e$, where b and e are the earliest and latest possible starting positions of *gap*(i, j), and then performing a bitwise "and" of the result and *precedes*(i, j). By the bitwise "or" we get all those keywords (i, j') that represent prefix matches already found and are in the correct distance from the current character position. The bitwise "and" finally computes from these keywords those that precede (i, j).

The function call *classes-match*(i, j, e) (to be explained in the next section) checks if the character classes in *gap*(i, j) (if any) match. The call returns true if the gap contains no character classes.

Algorithm 2. Procedure *traverse-output-path*(*state*).

```
$q \leftarrow state$
$traversed \leftarrow$ false
while not $traversed$ do
  for all elements $(i, j) \in output(q)$ do
    $b \leftarrow \max\{1, character\text{-}count - maxgap(i, j) - length(i, j)\}$
    $e \leftarrow character\text{-}count - mingap(i, j) - length(i, j)$
    if $e \geq 1$ then
      if ($j \in begins(i)$ or $precedes(i, j) \cap (\cup_{c=b}^{e} prefix\text{-}matches(i, c)) \neq \emptyset$)
      and $classes\text{-}match(i, j, e)$ then
        $insert\text{-}prefix\text{-}match(i, j, character\text{-}count)$
        if $j \in ends(i)$ then
          report a match of pattern $P_i$ at $character\text{-}count$
        end if
      end if
    end if
  end for
  if $q = initial\text{-}state$ then
    $traversed \leftarrow$ true
  else
    $q \leftarrow output\text{-}fail(q)$
  end if
end while
```

5 Matching for Patterns with Gaps and Character Classes

We now assume that the gaps in the regular patterns can also contain *character classes* such as in the gap "$.[B{-}H]..[AB]$", which matches any string *abcde*, where a, c and d are any characters, b is one of the characters B to H, and e is A or B. However, we require that the character classes, if any, appear in a fixed number of times and are in fixed positions in the gap, more specifically, in a fixed distance from the start of a minimum-length instance of the gap. This is always the case with a fixed-length gap, as with "$.[B{-}H]..[AB]$", where $\mathit{mingap}(i,j) = \mathit{maxgap}(i,j) = 5$ and the positions of the character classes are 2 and 5.

Gaps such as $[AB].\{l,h\}$ or $[AB]\{l,h\}$ with $l < h$ are not allowed, whereas a gap such as $.\{l,h\}[AB]$ is allowed (the position of $[AB]$ in the minimum-length gap instance is $l+1$). However, $[AB].\{l,h\}$ is a valid regular pattern of the form $g_1w_1g_2w_2$ with $g_1 = [AB]$, $w_1 = \epsilon$, $g_2 = .\{l,h\}$, $w_2 = \epsilon$.

We store all the character classes appearing in the patterns as bit vectors containing as many bits as there are characters in the entire character alphabet. Each distinct set is stored only once.

For each pattern P_i and each keyword number j we store a list $\mathit{classes}(i,j)$ containing pairs (c,C), where c is a character position in $\mathit{gap}(i,j)$ that contains a character class and C is a pointer to the bit vector representing the character class. The lists $\mathit{classes}(i,j)$ are arranged in an array of r entries, where r is the number of patterns in D. The entry for i is an array of k_i entries, where k_i is the number of keywords in pattern P_i. The entire array structure is of size $O(|D|)$ and, given (i,j), the beginning of the list $\mathit{classes}(i,j)$ can be accessed in constant time.

For example, if the gap-keyword pair (i,j) is $.[B{-}H]..[AB]w$, where w is the keyword, we have $\mathit{classes}(i,j) = \{(2,C_1),(5,C_2)\}$, where C_1 points to the class $[B{-}H]$ and C_2 points to the class $[AB]$. The character classes in the gap $[AB]\{3,3\}$ are represented by $\{(1,C),(2,C),(3,C)\}$, where C points to $[AB]$.

Given a matched keyword (i,j) and a character position e, the function $\mathit{classes\text{-}match}(i,j,e)$ (Alg. 3) tests whether or not the character classes in $\mathit{gap}(i,j)$ match with the gap starting at character position e.

Algorithm 3. Function *classes-match*(i,j,e).

```
for all $(c,C) \in classes(i,j)$ do
  if the input character at position $e+c-1$ does not belong to class $C$ then
    return false
  end if
end for
return true
```

6 Correctness and Complexity

The correctness of the algorithm follows from correct maintenance of the pattern-instance prefixes. We always create a new prefix, when a keyword (with a prescribed gap) starting a pattern is found, and whenever we find a new keyword (i, j) which correctly follows an already stored instance prefix (checked by *precedes*(i, j)), then a new instance prefix (i, j) is stored. If the newly found keyword (i, j) that correctly follows some instance prefix can also be the last keyword of an instance of pattern P_i, then we report a match.

The time needed for preprocessing was already discussed in Section 2. The main concern in the complexity analysis is the question of how many steps are performed for each scanned input character. For each new character the procedure *traverse-output-path* (Alg. 2) is executed, and thus we need to analyze how many times the outer **while** loop and the inner **for** loop are executed within one *traverse-output-path* call.

First notice that the number of iterations of the **while** loop in a call *traverse-output-path*(q) equals the length of the output path from state q. It is easy to see that this length equals the number of suffixes of *string*(q) that are keyword strings. When the **for** loop iterations will additionally be counted, we observe that the total number of their executions at character position c of the text T will be the number of keywords (i, j) that match at $T[c]$.

Checking whether or not $j \in$ *begins*(i) or $j \in$ *ends*(i) takes only time $O(1)$. Performing the bitwise "or" of the bit vectors returned by the function calls *prefix-matches*(i, c), $c = b, \ldots, e$ takes time $O((e - b + 1)(k_i/w))$, and performing the bitwise "and" of the result and the bit vector *precedes*(i, j) takes time $O(k_i/w)$. Checking the character-class matches in the function call *classes-match*$(i, j,$ *character-count*$)$ takes time $O(d)$, where d is the number of character classes in *gap*(i, j). Each of the procedure calls *prefix-matches*(i, c) and *insert-prefix-match*$(i, j,$ *character-count*$)$ takes time $O(\log r)$, for a dictionary of r patterns, because of the need to traverse the binary search tree for i. Thus we have:

Theorem 1. *Let D be a dictionary of size m containing r patterns that are regular expressions with bounded-length gaps and character classes in fixed positions in the gaps, and let T be a text of length n to be matched. After preprocessing in time $O(m + \sum_{i=1}^{r} k_i^2 \log k_i/w)$ and space $O(m + \sum_{i=1}^{r} k_i \log k_i)$, where k_i denotes the number of keywords in pattern P_i and w the number of bits in one memory word, the matching algorithm finds all occurrences in T of the patterns in D in time*

$$O(((d_1 + 1)(\log r + \frac{k}{w}) + d_2) \cdot \sum_{c=1}^{n}(K_c + 1)),$$

where $k = \max\{k_1, \ldots, k_r\}$, K_c is the number of keywords (i, j) matched at text position $T[c]$, d_1 is the maximal difference of the maximum and minimum lengths of a gap, and d_2 is the maximal number of character classes in a gap.

The workspace complexity of the algorithm is

$$O(maxdist \cdot K),$$

where maxdist is as defined in formula (2) and $K = \max\{K_c \mid c = 1, \ldots, n\}$.

Proof. The time bound follows from the above discussion. The workspace complexity simply follows from the fact that at most K new pairs (i, j) are inserted into the array *prefix-matches* at each character position, and that *maxdist* is the length of the array *prefix-matches.* As in the operating cycle (Alg. 1) *prefix-matches*$[(c-1) \bmod \mathit{maxdist} + 1]$ is set to empty, we conclude the space bound $O(\mathit{maxdist} \cdot K)$. □

The complexity bound of Theorem 1 is most interesting when only fixed-length wildcard gaps are present without any character classes, because then $d_1 = d_2 = 0$. Moreover, one may be willing to see an upper bound for K_c which is independent of text T and easy to determine from D.

For a keyword string y in D define

$$\mathit{closure}(y) = \{z \mid z \text{ is a keyword string and a suffix of } y\},$$

and for a set Y of keyword strings define

$$\mathit{occ\text{-}dictionary}(Y) = |\{(i, j) \mid y \in Y, y = \mathit{keyword}(i, j)\}|.$$

Using these definitions we replace K_c by an upper bound which only depends on D:

Corollary 2. *Let D be a dictionary of size m containing r regular-expression patterns with fixed-length wildcard gaps without character classes, and let T be a text of length n to be matched. After preprocessing in time $O(m + \sum_{i=1}^{r} k_i^2 \log k_i / w)$ and space $O(m + \sum_{i=1}^{r} k_i \log k_i)$ the matching algorithm finds all occurrences in T of the patterns in D in time*

$$O(n(\log r + \frac{k}{w})K_D),$$

where $K_D = \max\{\mathit{occ\text{-}dictionary}(\mathit{closure}(y)) \mid y \text{ is a keyword string in } D\}$. The workspace complexity of the algorithm is $O(\mathit{maxdist} \cdot K_D)$. □

7 The Case of a Single Pattern

From Corollary 2 we can conclude that in the case of a single regular-expression pattern P (with fixed-length wildcard gaps without character classes) we obtain for the matching algorithm the bound

$$O(n(\frac{k}{w} + K_D)),$$

where k is the number of keywords in P and K_D is as defined in Corollary 2 for $D = \{P\}$. This bound is better than what is obtained by Bille and Thorup [6] in the case $K_D < \log w$. This holds when for each keyword with number j in pattern P the number of keywords j' that are suffixes of keyword j is smaller than $\log w$. Recall the example given in the introduction suggesting that this condition often holds in practice.

In this section we will present an improvement upon the algorithm such that K_D can be replaced by a better upper bound. In the current algorithm the set *precedes*(i, j), denoted *precedes*(j) in the case of a single pattern, is computed as a preprocessing task separately for each j. In the `for` loop of Alg. 2 all keywords j in *output*(q) are separately checked as a candidate for continuing the already obtained matches of pattern prefixes. However, this checking could be done in one operation for all keywords in *output*(q) that are of the same length, provided that we would have computed the union of all *precedes* sets for these keywords. Let $j_i, j_2, \ldots, j_l$ be different keyword numbers such that $keyword(j_1) = keyword(j_2) = \ldots = keyword(j_l)$. During preprocessing we can easily construct, say, *precedes*(j_1) such that it contains the union *precedes*$(j_1) \cup \cdots \cup$ *precedes*(j_l), and delete from the output sets all numbers $j_2, \ldots, j_l$. After this change our algorithm will still work correctly, but now the time bound becomes

$$O(n(\frac{k}{w} + K'_D)),$$

where $K'_D = \max\{|closure(y)| \mid y \text{ is a keyword in } D\}$ for $D = \{P\}$. Notice that $K'_D \leq K_D$ and K'_D can be considerably smaller than K_D because now the different occurrences of the same keyword string are counted only once.

8 Discussion

Our new method for regular-expression-pattern matching can be considered as a generalization of some previous algorithms for string-pattern matching with gaps [4, 7]. In these algorithms—as in the algorithm of the present paper—the keywords in the patterns are recognized by using the Aho–Corasick automaton for multi-string matching. Our algorithm checks for the already found prefix matches of regular expression patterns whether or not the newly found keyword can be a legal continuation. The algorithm of Bille and Thorup [6] for regular expression matching is different from ours—though it also uses the keywords obtained from the expression and applies to keyword matching the Aho–Corasick automaton—in the sense that it still uses the simulation of a nondeterministic automaton, now constructed using the keyword numbers instead of single characters.

Although the algorithm of Bille and Thorup [6] has a good worst-case upper bound, it is hardly useful in our scenario of a possibly large number of regular expressions to be matched at the same time. If such a set is treated as a single regular expression, then the number k of different keyword occurrences in the expression, being a multiplier in the time bound, becomes very large compared to $\max\{k_1, \ldots, k_r\}$, which is a multiplier in our time bound, where k_i is the number of keyword occurrences in the ith expression.

Another possibility of using the algorithm of Bille and Thorup [6] in our scenario would be to apply the input text separately to all regular expression patterns. But then the Aho–Corasick automaton should be constructed for all expressions separately, and instead of feeding the input text once to one automaton it should be fed to r different automata, where r is the number of regular expression patterns. This would mean that the time bound of Bille and Thorup [6] would have an extra term nr instead of the term $n \log r$ in our bound. Using parallel computation the term nr could be made somewhat smaller, but not very much in the case of many thousands of expressions, as is typical in the applications.

Our aim in the near future is to perform experimental analysis of our algorithm. Preliminary experiments have already been conducted with the algorithm that was a direct extension of our algorithm for string-pattern matching with gaps [7]. These were very promising, even though that extension was not at all as good as the present one as regards the worst-case time complexity.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. Bille, P.: New algorithms for regular expression matching. In: Proc. of the 33rd Internat. Colloq. Automata, Languages and Programming, pp. 643–654 (2006)
3. Bille, P., Farach-Colton, M.: Fast and compact regular expression matching. Theor. Comput. Sci. 409(3), 486–496 (2008)
4. Bille, P., Gørtz, I.L., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. Theor. Comput. Sci. 443, 25–34 (2012)
5. Bille, P., Thorup, M.: Faster regular expression matching. In: Proc. of the 36th Internat. Colloq. Automata, Languages and Programming, pp. 171–182 (2009)
6. Bille, P., Thorup, M.: Regular expression matching with multi-strings and intervals. In: Proc. of the 21st Annual ACM-SIAM Symp. on Discrete Algorithms, pp. 1297–1308 (2010)
7. Haapasalo, T., Silvasti, P., Sippu, S., Soisalon-Soininen, E.: Online Dictionary Matching with Variable-Length Gaps. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 76–87. Springer, Heidelberg (2011)
8. Myers, E.W.: A four russians algorithm for regular expression pattern matching. J. ACM 39(2), 430–448 (1992)
9. Schnitger, G.: Regular expressions and NFAs without ϵ-transitions. In: Proc. of the 23rd Annual Symp. on Theoretical Aspects of Computer Science, pp. 432–443 (2006)
10. Sen, S., Spatscheck, O., Wang, D.: Accurate, scalable in-network identification of p2p traffic using application signatures. In: Proc. of the 13th Internat. Conf. on World Wide Web, pp. 512–521 (2004)
11. Thompson, K.: Regular expression search algorithm. Commun. ACM 11(6), 419–422 (1968)

Infiniteness and Boundedness in 0L, DT0L, and T0L Systems

Tim Smith

College of Computer and Information Science, Northeastern University
Boston, MA 02115, USA
smithtim@ccs.neu.edu

Abstract. We investigate the boundary between finiteness and infiniteness in three types of L systems: 0L, DT0L, and T0L. We establish necessary and sufficient conditions for 0L, DT0L, and T0L systems to be infinite, and characterize the boundedness of finite classes of such systems. First, we give a pumping lemma for these systems, proving that the language of a system is infinite iff the system is pumpable. Next, we show that the number of steps needed to derive any string in any finite 0L or DT0L system is bounded by a function depending only on the size of the alphabet, and not on the production rules or start string. This alphabet boundedness does not hold for finite T0L systems in general. Finally, we show that every infinite 0L system has an infinite D0L subsystem.

1 Introduction

L systems are parallel rewriting systems which were originally introduced to model growth in simple multicellular organisms. With applications in biological modelling, fractal generation, and artificial life, L systems have given rise to a rich body of research [6,2]. L systems can be restricted and generalized in various ways, yielding a hierarchy of language classes.

The simplest L systems are D0L systems (deterministic Lindenmayer systems with 0 symbols of context), in which a morphism is successively applied to a start string or "axiom". In [7], Vitányi gives a necessary and sufficient condition under which a D0L system is finite, and gives an upper bound on the size of a finite D0L language in terms of the size of the alphabet.

Two well-studied generalizations of D0L systems are 0L systems, which introduce nondeterminism by changing the morphism to a finite substitution, and DT0L systems, in which the morphism is replaced by a set of morphisms or "tables". Generalizing in both directions at once yields the class of T0L systems. Figure 1 depicts the inclusions among these classes. We extend Vitányi's work to these systems.

First, we provide a necessary and sufficient condition under which a T0L system is infinite, in the form of a pumping lemma. In getting this result, we adapt a proof technique used in [5] to obtain a pumping lemma for ET0L systems. It follows from our pumping lemma that every infinite T0L language has an infinite D0L subset.

Next, we look for upper bounds on finite 0L, DT0L, and T0L systems in terms of alphabet size. In contrast to D0L systems, there is no upper bound on the size of a finite 0L or DT0L language in terms of the size of the alphabet alone. However, we show that there is such a bound on the number of steps needed to derive a string in any finite 0L or DT0L system. For finite T0L systems in general, a counterexample shows that no such alphabet-only bound holds. Figure 2 summarizes these results.

Finally, we consider the notion of a D0L subsystem of a 0L system, formed by choosing a single production for each symbol from the finite substitution. We show that every infinite 0L system has an infinite D0L subsystem; this constitutes a necessary and sufficient condition for a 0L system to be infinite. We also consider the notion of a D0L subsystem of a DT0L system, formed by choosing a single table from the set of tables. A simple counterexample shows that not every infinite DT0L system has an infinite D0L subsystem.

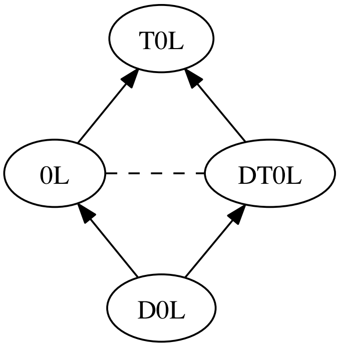


Fig. 1. Inclusion diagram. Arrows indicate proper inclusion of the lower class by the upper class; the dashed line indicates incomparability.

	alphabet size-bounded?	alphabet step-bounded?
D0L	yes	yes
0L	no	yes
DT0L	no	yes
T0L	no	no

Fig. 2. Alphabet boundedness of finite D0L, 0L, DT0L, and T0L systems

Related Work. Finiteness of all the L systems considered in this paper is decidable from Theorem 4.1 of [2]. That the size of the alphabet bounds the number of steps needed to derive λ in a 0L system was known from Lemma 1.3 of [6]; for finite 0L systems, our Theorem 15 generalizes this result to include non-empty strings.

Nishida [3] investigated "quasi-deterministic" 0L systems, those for which there is an integer C such that the cardinality of the set of strings generated in exactly n steps is less than C for every n. Nishida and Salomaa [4] investigated "slender" 0L languages, those for which there is a constant k such that the language has at most k strings of any given length.

Corollary 5, which states our pumping lemma for DT0L systems, can also be proved via a connection with non-negative integer matrices. Each table in a DT0L system can be associated with a "growth matrix" indicating for each production, how many times each symbol appears on the righthand side of that production. Jungers et al. [1] consider the "joint spectral radius" ρ of a finite set of such matrices, distinguishing four cases. In cases (1) and (2) ($\rho = 0$ or $\rho = 1$ with bounded products), the associated DT0L system is finite, whereas in cases (3) and (4) ($\rho > 1$ or $\rho = 1$ with unbounded products), by their Corollary 1 and Proposition 2, assuming every symbol is reachable, the system is pumpable.

Outline of Paper. The paper is organized as follows. Section 2 gives preliminary definitions. Section 3 presents our pumping lemma for T0L systems. Section 4 examines alphabet boundedness for finite 0L, DT0L, and T0L systems. Section 5 studies D0L subsystems of 0L and DT0L systems. Section 6 gives our conclusions.

2 Definitions

An **alphabet** A is a finite set of symbols. A **string** is an element of A^*. λ denotes the empty string. A **language** is a subset of A^*. A **morphism** on A is a map h from A^* to A^* such that for all $x, y \in A^*$, $h(xy) = h(x)h(y)$. Notice that $h(\lambda) = \lambda$. h is **nonerasing** if for every $c \in A$, $h(c) \neq \lambda$. A **finite substitution** on A is a map σ from A^* to 2^{A^*} such that (1) for all $x \in A^*$, $\sigma(x)$ is finite and nonempty, and (2) for all $x, y \in A^*$, $\sigma(xy) = \{x'y' \mid x'$ is in $\sigma(x)$ and y' is in $\sigma(y)\}$. Notice that $\sigma(\lambda) = \{\lambda\}$. For a language L, we define $\sigma(L) = \{x' \mid x'$ is in $\sigma(x)$ for some $x \in L\}$.

A **D0L system** is a tuple $G = (A, h, w)$ where A is an alphabet, h is a morphism on A, and w is in A^*. For $x, y \in A^*$ and $i \geq 0$, we write $x \xrightarrow{i} y$ iff $h^i(x) = y$.

A **0L system** is a tuple $G = (A, \sigma, w)$ where A is an alphabet, σ is a finite substitution on A, and w is in A^*. For $x, y \in A^*$ and $i \geq 0$, we write $x \xrightarrow{i} y$ iff $\sigma^i(x) \ni y$.

A **DT0L system** is a tuple $G = (A, H, w)$ where A is an alphabet, H is a finite nonempty set of morphisms on A (called "tables"), and w is in A^*. For $x, y \in A^*$ and $i \geq 0$, we write $x \xrightarrow{i} y$ iff $h_k \cdots h_1(x) = y$ for some $h_1, \ldots, h_k \in H$.

A **T0L system** is a tuple $G = (A, T, w)$ where A is an alphabet, T is a finite nonempty set of finite substitutions on A (called "tables"), and w is in A^*. For $x, y \in A^*$ and $i \geq 0$, we write $x \xrightarrow{i} y$ iff $\sigma_k \cdots \sigma_1(x) \ni y$ for some $\sigma_1, \ldots, \sigma_k \in T$.

For any of the above systems G, w is called the "axiom" or "start string". The language of G is $L(G) = \{s \mid w \xrightarrow{i} s$ for some $i \geq 0\}$. Call G finite iff $L(G)$ is finite. Intuitively, a derivation in G means a sequence of steps, starting with w unless otherwise specified, each consisting of a string together with the precise table and/or productions used to derive it from the previous step. For formal definitions, see [6]. **D0L**, **0L**, **DT0L**, and **T0L** are the classes of D0L, 0L, DT0L, and T0L languages, respectively. Clearly D0L $\subseteq$ 0L $\subseteq$ T0L and D0L $\subseteq$ DT0L $\subseteq$ T0L. In fact, 0L and DT0L are incomparable, making all of these inclusions proper [6].

A D0L (0L, DT0L, T0L) system G with axiom w is **step-bounded by** n iff for every $s \in L(G)$, there is an $m \leq n$ such that $w \xrightarrow{m} s$. Let C be any class of D0L (0L, DT0L, T0L) systems. C is **alphabet size-bounded** iff for every alphabet A, there is an $n \geq 0$ such that for every $G \in C$ for which the alphabet of G is A, $|L(G)| \leq n$. C is **alphabet step-bounded** iff for every alphabet A, there is an $n \geq 0$ such that for every $G \in C$ for which the alphabet of G is A, G is step-bounded by n. Clearly if C is alphabet-size bounded, then C is alphabet step-bounded, since the same n will suffice.

3 Pumping Lemma for T0L Systems

A T0L system $G = (A, T, w)$ is **pumpable** iff there are $x, y \in A$ such that (1) some $s_0 \in L(G)$ contains x, and (2) for some composition t of tables from T, $t(x)$ includes a string s_1 containing distinct occurrences of x and y and $t(y)$ includes a string s_2 containing y.

Lemma 1. *Suppose the T0L system $G = (A, T, w)$ is pumpable. Then $L(G)$ is infinite.*

Proof. Since s_0 is in $L(G)$ and t is a composition of tables from T, $t^i(s_0) \subseteq L(G)$ for every $i \geq 0$. A simple induction shows that for all $i \geq 0$, $t^i(s_0)$ includes a string containing x and at least i copies of y. Hence $L(G)$ is infinite. □

Lemma 2. *Suppose the T0L system $G = (A, T, w)$ is infinite. Then G is pumpable.*

Proof. We assume a familiarity with [5], particularly the notions of an ET0L system, derivation tree, marked node, and branch node. G can be treated as an ET0L system in which the alphabet and terminal alphabet are identical. Following the proof of Theorem 15 in [5], for any node in a derivation tree, consider the "marked set", or set of marked symbols which appear on the same level of the tree. As shown in that proof, since G is infinite, there is an $x \in A$ such that some derivation tree in G of a string in which every position is marked has a path with two branch nodes labelled by x, with one an ancestor of the other, with the same marked set. Call the strings in which the ancestor and descendant nodes appear w_1 and w_2, respectively. Let t be the composition of tables which was applied to w_1 to derive w_2.

Since the ancestor node labelled by x in w_1 is a branch node, its descendant string in w_2 contains, in addition to the descendant node labelled by x, a marked node labelled by some $e \in A$. Now, since w_1 and w_2 have the same marked set, every $c \in A$ which labels a marked node in w_2 also labels a marked node in w_1. By definition, every marked node in w_1 has a marked descendant in w_2. A simple induction then shows that for every $i \geq 0$, there is a $c \in A$ such that w_2 contains a marked node labelled by c, and some $s \in t^i(e)$ contains c. Hence for every $i \geq 0$, $t^i(e)$ contains a non-empty string. So there are $j \geq 0$, $k \geq 1$ and $y \in A$ such that $t^j(e)$ includes a string containing y and $t^k(y)$ includes a string containing y. Then since $t(x)$ includes a string containing distinct occurrences of x and e, $t^{j+1}(x)$ includes a string containing distinct occurrences of x and y. Then $t^{k(j+1)}(x)$ includes a string containing distinct occurrences of x and y and $t^{k(j+1)}(y)$ includes a string containing y. So G is pumpable. □

Theorem 3. *A T0L system is infinite iff it is pumpable.*

Proof. Immediate from Lemmas 1 and 2. □

Corollary 4. *A 0L system $G = (A, \sigma, w)$ is infinite iff there are $x, y \in A$ such that (1) some $s \in L(G)$ contains x, and (2) for some $i \geq 0$, $\sigma^i(x)$ includes a string containing distinct occurrences of x and y and $\sigma^i(y)$ includes a string containing y.*

Corollary 5. *A DT0L system $G = (A, H, w)$ is infinite iff there are $x, y \in A$ such that (1) some $s \in L(G)$ contains x, and (2) for some composition h of morphisms from H, $h(x)$ contains distinct occurrences of x and y and $h(y)$ contains y.*

Corollary 6. *Every infinite T0L language has an infinite D0L subset.*

Proof. Take any infinite T0L language L with T0L system $G = (A, T, w)$. By Theorem 3, G is pumpable. Let h be a morphism on A such that $h(x) = s_1$, $h(y) = s_2$ unless $x = y$, and for every other $c \in A$, $h(c) = s$ for some $s \in t(c)$. Then the language of the D0L system (A, h, s_0) is an infinite subset of L. □

4 Alphabet Boundedness

In this section we examine the alphabet size-boundedness and step-boundedness of 0L, DT0L, and T0L systems. For D0L, Corollary 4 of [7] implies the following.

Theorem 7 (Vitányi). *The class of finite D0L systems is alphabet size-bounded and alphabet step-bounded.*

4.1 0L

We first give a simple counterexample to show that the class of finite 0L systems is not alphabet size-bounded.

Theorem 8. *The class of finite 0L systems is not alphabet size-bounded.*

Proof. Let $A = \{\mathtt{a}, \mathtt{b}\}$ and take any $n \geq 0$. Let $w = \mathtt{a}$. Let σ be a finite substitution on A such that $\sigma(\mathtt{a}) = \{\mathtt{b}, \mathtt{bb}, \mathtt{bbb}, \ldots, \mathtt{b}^n\}$ and $\sigma(\mathtt{b}) = \{\mathtt{b}\}$. Let $G = (A, \sigma, w)$. Then $L(G) = \{\mathtt{a}, \mathtt{b}, \mathtt{bb}, \mathtt{bbb}, \ldots, \mathtt{b}^n\}$. So $L(G)$ is finite, but $|L(G)| > n$. So the class of finite 0L systems is not alphabet size-bounded. □

Next we will show that the class of finite 0L systems is alphabet step-bounded. We begin with some definitions. Take any 0L system (A, σ, w) and any $c \in A$. For any $s \in A^*$, c is **reachable from** s iff for some $i \geq 0$, $\sigma^i(s)$ includes a string which contains c. c is **reachable** iff c is reachable from w. Let $L(s)$ be the language of the 0L system (A, σ, s). c is **mortal** (c is in M) iff $\sigma^i(c) = \{\lambda\}$ for some $i \geq 0$. c is **vital** (c is in V) iff c is not in M. c is **recursive** (c is in R) iff c is reachable from some $s \in \sigma(c)$. c is **monorecursive** (c is in MR) iff for every

s such that for some $i \geq 0$, $\sigma^i(c)$ includes s and s contains c, s is in M^*cM^*. For each $i \geq 0$, let $reach_c(i) = \{s \in \sigma^i(c) \mid c \text{ is reachable from } s\}$.

We now build up a series of lemmas toward our result that the class of finite 0L systems is alphabet step-bounded. Lemmas 9 and 10 are given without proof but are not difficult to verify.

Lemma 9. *Suppose c is in $R - MR$. Then $L(c)$ is infinite.*

Lemma 10. *For all $c \in M$, $\sigma^{|M|}(c) = \{\lambda\}$.*

Lemma 11. *Suppose c is in MR and s is in $reach_c(i)$ for some i. Then s is in M^*dM^* for some $d \in MR$.*

Proof. Since c is reachable from s, s must contain at least one symbol in V. s cannot contain more than one symbol in V, otherwise c would not be in MR. So s contains exactly one symbol d in V. Since c is reachable from d and d is reachable from c, d is in R. Now suppose d is in $R - MR$. Then $L(d)$ contains a string s' which includes d and a symbol in V. Since c is reachable from d, $L(s')$ contains a string which includes c and a symbol in V. Then $L(d)$ contains such a string. But then $L(c)$ contains such a string, a contradiction, since c is in MR. So d is in MR. Then s is in M^*dM^*. □

Lemma 12. *Suppose c is in MR. Then there is a k such that $1 \leq k \leq |MR|$ and some string in $\sigma^k(c)$ contains c.*

Proof. Since c is in MR, there is a $k \geq 1$ such that some string in $\sigma^k(c)$ contains c. Take the smallest such k. Then there is an s in M^*cM^* and derivation D of s from c in k steps. Suppose $k > |MR|$. Then the c in s has $> |MR|$ ancestors in D. Take any such ancestor d. c is reachable from d, so by Lemma 11, since d is not in M, d is in MR. So every ancestor of the c in s is in MR. But then one such ancestor must repeat, and the derivation could have been shortened to yield a k' such that $1 \leq k' < k$. Therefore $k \leq |MR|$. □

Lemma 13. *Suppose c is in A and $L(c)$ is finite. Then there is a k such that $1 \leq k \leq |A|$ and $reach_c(|A|^2) = reach_c(|A|^2 + k)$.*

Proof. Suppose c is not in MR. Then by Lemma 9, c is not in R. So c is not reachable from any s in $\sigma(c)$. Then for every $i \geq 1$, $reach_c(i) = \{\}$. So say c is in MR. From Lemma 12, there is a k such that $1 \leq k \leq |MR|$ and some $s \in \sigma^k(c)$ contains c. Then s is in $reach_c(k)$. Let $Set(i) = \{a \in MR \mid \text{some string in } reach_c(i) \text{ includes } a\}$. Take any $i \geq 0$ and $a \in Set(ki)$. Some $s' \in reach_c(ki)$ includes a. Then s' is in $\sigma^{ki}(c)$. Since s contains c, s' is a substring of some s'' in $\sigma^{ki}(s)$. s'' is in $\sigma^{k(i+1)}(c)$. Since c is reachable from s', c is reachable from s''. So s'' is in $reach_c(k(i+1))$. Hence a is in $Set(k(i+1)$. So for all $i \geq 0$, $Set(ki)$ is a subset of $Set(k(i+1))$. Hence there is an $i < |MR|$ such that $Set(ki) = Set(k(i+1))$. Let $m = ki + |M|$ and $n = k(i+1) + |M|$. We will show that $reach_c(m) = reach_c(n)$.

Take any $s \in reach_c(m)$. There is some $s' \in \sigma^{ki}(c)$ such that s is in $\sigma^{|M|}(s')$. Then s' is in $reach_c(ki)$, so by Lemma 11, s' is in M^*dM^* for some $d \in MR$.

Then by Lemma 10, $\sigma^{|M|}(d)$ includes s. Now d is in $Set(ki)$, hence in $Set(k(i+1))$. Then by Lemma 11, $reach_c(k(i+1))$ contains an $s'' \in M^*dM^*$. So s is in $\sigma^{|M|}(s'')$, hence in $\sigma^{k(i+1)+|M|}(c)$, hence in $reach_c(n)$.

Now take any $s \in reach_c(n)$. There is some $s' \in \sigma^{k(i+1)}(c)$ such that s is in $\sigma^{|M|}(s')$. Then s' is in $reach_c(k(i+1))$, so by Lemma 11, s' is in M^*dM^* for some $d \in MR$. Then by Lemma 10, $\sigma^{|M|}(d)$ includes s. Now d is in $Set(k(i+1))$, hence in $Set(ki)$. Then by Lemma 11, $reach_c(ki)$ contains an $s'' \in M^*dM^*$. So s is in $\sigma^{|M|}(s'')$, hence in $\sigma^{ki+|M|}(c)$, hence in $reach_c(m)$.

Therefore $reach_c(m) = reach_c(n)$. Then for all $i \geq m$, $reach_c(i) = reach_c(i+k)$. Then since $m \leq |MR|\cdot(|MR|-1)+|M| \leq |A|^2$, $reach_c(|A|^2) = reach_c(|A|^2+k)$. □

Theorem 14. *For every alphabet A, there are $f \geq 1, g \geq 0$ such that for every finite 0L system (A, σ, w), $\sigma^g(w) = \sigma^{g+f}(w)$.*

Proof. Let $f(0) = 1$ and for every $x \geq 1$, $f(x) = x!f(x-1)$. Let $g(0) = 0$ and for every $x \geq 1$, $g(x) = x^2 + g(x-1) + f(x)$. Take any finite 0L system $G = (A, \sigma, w)$. We will show by induction on $|A|$ that $\sigma^{g(|A|)}(w) = \sigma^{g(|A|)+f(|A|)}(w)$.

Take the base case of $|A| = 0$. Then $w = \lambda$. Then for all $i \geq 0$, $\sigma^i(w) = \{\lambda\}$. So $\sigma^{g(0)}(w) = \sigma^{g(0)+f(0)}(w)$.

So say $|A| \geq 1$ and $w \neq \lambda$. Suppose for induction that for every finite 0L system (A', σ', w') such that $|A'| < |A|$, $\sigma'^{g(|A'|)}(w') = \sigma'^{g(|A'|)+f(|A'|)}(w')$. Take any c in w. By Lemma 13, there is a k' such that $1 \leq k' \leq |A|$ and $reach_c(|A|^2) = reach_c(|A|^2 + k')$. Let $k = |A!|$ and $t = |A|^2$. Then since k is divisible by k', $reach_c(t) = reach_c(t+k)$. Let $x = f(|A|-1)$ and $y = g(|A|-1)$. We will show that $\sigma^{t+y+kx}(c) = \sigma^{t+y+2kx}(c)$.

Take any $s \in \sigma^{t+y+kx}(c)$. Then there is an $r \in \sigma^t(c)$ such that s is in $\sigma^{y+kx}(r)$. Suppose c is reachable from r. Then r is in $reach_c(t)$. Since $reach_c(t) = reach_c(t+kx)$, r is in $\sigma^{t+kx}(c)$. Then s is in $\sigma^{t+y+2kx}(c)$. So say c is not reachable from r. Then by the induction hypothesis, $\sigma^y(r) = \sigma^{y+x}(r)$. Hence $\sigma^{y+kx}(r) = \sigma^{y+2kx}(r)$. Then s is in $\sigma^{y+2kx}(r)$. So s is in $\sigma^{t+y+2kx}(c)$.

Now take any $s \in \sigma^{t+y+2kx}(c)$. Then there is an $r \in \sigma^{t+kx}(c)$ such that s is in $\sigma^{y+kx}(r)$. Suppose c is reachable from r. Then r is in $reach_c(t+kx)$. Since $reach_c(t+kx) = reach_c(t)$, r is in $\sigma^t(c)$. Then s is in $\sigma^{t+y+kx}(c)$. So say c is not reachable from r. Then by the induction hypothesis, $\sigma^y(r) = \sigma^{y+x}(r)$. Hence $\sigma^y(r) = \sigma^{y+kx}(r)$. Then s is in $\sigma^y(r)$. So s is in $\sigma^{t+y+kx}(c)$.

So for all c in w, $\sigma^{t+y+kx}(c) = \sigma^{t+y+2kx}(c)$. Hence $\sigma^{t+y+kx}(w) = \sigma^{t+y+2kx}(w)$. Now $t+y+kx = g(|A|)$ and $kx = f(|A|)$, completing the induction. □

Theorem 15. *The class of finite 0L systems is alphabet step-bounded.*

Proof. Take any alphabet A. Take any f, g meeting the conditions of Theorem 14 for A. Let $n = f + g$. Take any finite 0L system $G = (A, \sigma, w)$ and any $s \in L(G)$. Then there is a lowest $i \geq 0$ such that s is in $\sigma^i(w)$. Suppose $i > n$. By Theorem 14, $\sigma^g(w) = \sigma^{g+f}(w)$. Then $\sigma^i(w) = \sigma^{i-f}(w)$. Then s is in $\sigma^{i-f}(w)$, a contradiction. So G is step-bounded by n. Hence the class of finite 0L systems is alphabet step-bounded. □

4.2 DT0L

In this subsection, we first give a simple counterexample to show that the class of finite DT0L systems is not alphabet size-bounded. We then show that this class is alphabet step-bounded, first proving a lemma about a more restricted class of systems.

Theorem 16. *The class of finite DT0L systems is not alphabet size-bounded.*

Proof. Let $A = \{\mathtt{a}, \mathtt{b}\}$ and take any $n \geq 0$. Let $w = \mathtt{a}$. For every $1 \leq i \leq n$, let h_i be a morphism on A such that $h_i(\mathtt{a}) = \mathtt{b}^i$ and $h_i(\mathtt{b}) = \mathtt{b}$. Let $H = \{h_1, \dots, h_n\}$. Let $G = (A, H, w)$. Then $L(G) = \{\mathtt{a}, \mathtt{b}, \mathtt{bb}, \mathtt{bbb}, \dots, \mathtt{b}^n\}$. So $L(G)$ is finite, but $|L(G)| > n$. So the class of finite DT0L systems is not alphabet size-bounded. □

Take any DT0L system $G = (A, H, w)$. For any $s \in A^*$, let $L(s)$ be the language of the DT0L system (A, H, s). If for every $h \in H$, h is nonerasing, G is called a propagating DT0L system or **PDT0L system** [2].

Lemma 17. *For every alphabet A, there is an $m \geq 0$ such that for every finite PDT0L system $G = (A, H, w)$, for any $h_1, \dots, h_n \in H$ such that $n > m$, there are j, k such that $0 \leq j < k \leq n$ and $h_j \cdots h_1(w) = h_k \cdots h_1(w)$.*

Proof. Take any alphabet A. Take any $m > (1 + (|A| + 1)!) \cdot |A|^{|A|}$. Take any finite PDT0L system $G = (A, H, w)$. For any $A' \subseteq A$, call $S = \{s_0, s_1, s_2, \dots, s_n\}$ relevant to A' if every s_i is in A'^*, $L(s_0)$ is finite, and for every $1 \leq i \leq n$, there is an $h \in H$ such that $h(s_{i-1}) = s_i$. Notice that since G is a PDT0L system, for every i, $|s_i| \leq |s_{i+1}|$. Let $\mathit{Jumps}(A', S) = |\{i \mid |s_i| < |s_{i+1}|\}|$. We will show by induction on $|A|$ that for every S relevant to A, $\mathit{Jumps}(A, S) \leq (|A| + 1)!$. Take any $S = \{s_0, s_1, s_2, \dots, s_n\}$ relevant to A.

For the base case, suppose $|A| = 0$. Then for every $s_i \in S$, $s_i = \lambda$. So $\mathit{Jumps}(A, S) = 0$.

Now suppose for induction that for every $A' \subsetneq A$, for every S' relevant to A', $\mathit{Jumps}(A', S') \leq (|A'| + 1)!$. If $s_0 = \lambda$, clearly $\mathit{Jumps}(A, S) = 0$. So say $s_0 \neq \lambda$. Take any c in s_0. Let $S' = \{s'_0, s'_1, s'_2, \dots, s'_n\}$, where $s'_0 = c$ and each s'_i is the descendant string in s_i of the c in s_0. We will show that $\mathit{Jumps}(A, S') \leq 1 + |A|!$. If there is no j such that $|s'_j| < |s'_{j+1}|$, then $\mathit{Jumps}(A, S') = 0$. So say there is such a j. Take the first such j. Then $s'_j = d$ for some $d \in A$, since $|s'_0| = 1$. Suppose there is a $k > j$ such that s'_k contains d. Then there is a composition h of morphisms from H such that $h(d)$ contains d and $|h(d)| > 1$. Then for every $i \geq 0$, $|h^i(d)| > i$. But then $L(d)$ is infinite, hence $L(s_0)$ is infinite, a contradiction. So for every $k > j$, s'_k does not contain d. Let $S'' = \{s'_{j+1}, s'_{j+2}, \dots, s'_n\}$. Then $\mathit{Jumps}(A, S') = 1 + \mathit{Jumps}(A - d, S'')$. So by the induction hypothesis, $\mathit{Jumps}(A, S') \leq 1 + (|A| - 1 + 1)! \leq 1 + |A|!$. Now for each c in s_0, there will be some S' constructed in this way. For any two occurrences of the same c, S' will be the same. Then there are at most $|A|$ distinct S's. Therefore $\mathit{Jumps}(A, S) \leq |A| \cdot (1 + |A|!) \leq (|A| + 1)!$, completing the induction.

Now take any $h_1, \ldots, h_n \in H$ such that $n > m$. For each $0 \leq i \leq n$, let $s_i = h_i \cdots h_1(w)$. Then $s_0 = w$. So $S = \{s_0, s_1, s_2, \ldots, s_n\}$ is relevant to A. Hence $\mathit{Jumps}(A, S) \leq (|A| + 1)!$. Then since $n > m$, there are j', k' such that $|s_{j'}| = |s_{k'}|$ and $j' + |A|^{|A|} \leq k$. Then every s_i between $s_{j'}$ and $s_{k'}$ has the same length. But at most $|A|^{|A|}$ such s_i are distinct, since each c in $s_{j'}$ has only one descendant in each s_i, and any two occurrences of the same c have the same descendant. So there are j, k such that $0 \leq j' \leq j < k \leq k' \leq n$ and $s_j = s_k$, which was to be shown. □

Theorem 18. *For every alphabet A, there is a $b \geq 0$ such that for every finite DT0L system $G = (A, H, w)$, for any $h_1, \ldots, h_n \in H$ such that $n > b$, there are p, q such that $1 \leq p \leq q \leq n$ and $h_n \cdots h_{q+1} h_{p-1} \cdots h_1(w) = h_n \cdots h_1(w)$.*

Proof. Take any alphabet A. Take any m meeting the conditions of Lemma 17 for A. Take any $b > m \cdot 2^{|A|}$. Take any finite DT0L system $G = (A, H, w)$. Take any $h_1, \ldots, h_n \in H$ such that $n > b$. Let $s = h_n \cdots h_1(w)$. For each $0 \leq i \leq n$, let $s_i = h_i \cdots h_1(w)$. Then $s_0 = w$ and $s_n = s$. For each $0 \leq i < n$, let $f_i = h_n \cdots h_{i+1}$ and let $\mathit{Stay}_i = \{c \in A \mid f_i(c) \neq \lambda\}$. Each Stay_i is one of $2^{|A|}$ possible sets. Then since $n > m \cdot 2^{|A|}$, there is a subset Stay of A and $0 \leq z_0 < z_1 < z_2 < \cdots < z_m < n$ such that for every $0 \leq i \leq m$, $\mathit{Stay}_{z_i} = \mathit{Stay}$. Let $\mathit{Gone} = A - \mathit{Stay}$. Notice that for every $c \in \mathit{Gone}$ and $1 \leq i \leq n$, $h_i(c)$ is in Gone^*. For each $x \in A^*$, let $\mathit{Core}(x)$ be the string obtained from x by erasing all occurrences of symbols in Gone.

We now construct a finite PDT0L system G'. Let $A' = \mathit{Stay}$ and $w' = \mathit{Core}(s_{z_0})$. For each $1 \leq i \leq m$ and $c \in \mathit{Stay}$, let $h'_i(c) = \mathit{Core}(h_{z_i} \cdots h_{z_{i-1}+1}(c))$. Let $H' = \{h'_1, \ldots, h'_m\}$. Now, for any $c \in \mathit{Stay}$ and any $h'_i \in H'$, clearly $h'_i(c)$ is in Stay^*. Further, since $f_{z_{i-1}}(c) \neq \lambda$, $h'_i(c) \neq \lambda$. Therefore $G' = (A', H', w')$ is a PDT0L system. Further, since $L(G)$ is finite, and w' was obtained by erasing letters from a string in $L(G)$, and each h'_i was obtained by composing tables from H and erasing letters from the result, $L(G')$ is finite.

Now for each $0 \leq i \leq m$, let $s'_i = t'_i \cdots t'_1(w)$. Then by Lemma 17, there are j, k such that $0 \leq j < k \leq m$ and $s'_j = s'_k$. Notice that for each $0 \leq i \leq m$, $s'_i = \mathit{Core}(s_{z_i})$. Then $\mathit{Core}(s_{z_j}) = \mathit{Core}(s_{z_k}) = s'_j = s'_k$. Now $f_{z_k}(s_{z_k}) = s$. Therefore $f_{z_k}(\mathit{Core}(s_{z_k})) = s$. Then $f_{z_k}(\mathit{Core}(s_{z_j})) = s$. So then $f_{z_k}(s_{z_j}) = s$. So set $p = z_j + 1$ and $q = z_k$. Then $h_n \cdots h_{q+1} h_{p-1} \cdots h_1(w) = h_n \cdots h_{q+1}(s_{z_j})$ $= f_{z_k}(s_{z_j}) = s$, as desired. □

Theorem 19. *The class of finite DT0L systems is alphabet step-bounded.*

Proof. Take any alphabet A. Take any b meeting the conditions of Theorem 18 for A. Take any finite DT0L system $G = (A, H, w)$. Then by Theorem 18, any derivation in G with more than b steps can be shortened. So G is step-bounded by b. Hence the class of finite DT0L systems is alphabet step-bounded. □

4.3 T0L

Theorem 20. *The class of finite T0L systems is not alphabet step-bounded.*

Proof. Let $A = \{\mathtt{a}, \mathtt{b}, \mathtt{x}\}$ and take any $n \geq 0$. Let $w = (\mathtt{ax})^{n+1}$. For every $1 \leq i \leq n+1$, let σ_i be a finite substitution on A such that $\sigma_i(\mathtt{a}) = \{\mathtt{a}, \mathtt{b}^i\}$, $\sigma_i(\mathtt{b}) = \{\mathtt{b}\}$, and $\sigma_i(\mathtt{x}) = \{\mathtt{x}\}$. Let $T = \{\sigma_1, \ldots, \sigma_{n+1}\}$. Let $G = (A, T, w)$. Clearly G is finite. Let $s = \mathtt{bxbbxbbbx} \cdots \mathtt{b}^{n+1}\mathtt{x}$. Then s can be derived from w in $n+1$ steps, by applying each table in turn to replace an $\mathtt{a}$ by $\mathtt{b}$s. In any derivation of s from w, at each step, at most one $\mathtt{a}$ can be replaced by $\mathtt{b}$s, otherwise s would become unreachable. So at least $n+1$ steps are needed to derive s. Hence G is not step-bounded by n. So the class of finite T0L systems is not alphabet step-bounded. □

Corollary 21. *The class of finite T0L systems is not alphabet size-bounded.*

5 D0L Subsystems

By Corollary 6, every infinite T0L language has an infinite D0L subset. In this section, we consider a related notion, that of a D0L subsystem of a 0L or DT0L system. Such a subsystem not only generates a subset of the original language, but also shares structural characteristics with the original system.

5.1 0L

Let $G = (A, \sigma, w)$ be a 0L system. A **D0L subsystem** of G is a D0L system $G' = (A, h, w)$ such that for every $c \in A$, $h(c)$ is in $\sigma(c)$. Notice that $L(G') \subseteq L(G)$.

Lemma 22. *Take any 0L system $G = (A, \sigma, w)$ with mortal symbols M and vital symbols V. Take any D0L subsystem $G' = (A, h, w)$ of G such that for every $c \in V$, $h(c)$ contains some $d \in V$. Let G' have mortal symbols M' and vital symbols V'. Then $M' = M$ and $V' = V$.*

Proof. Take any $c \in M$. Then $\sigma^i(c) = \{\lambda\}$ for some $i \geq 0$. Then since G' is a D0L subsystem of G, $h^i(c) = \lambda$. So c is in M'. Now take any $c \in V$. We will show that c is in V' by induction on the number n of symbols which are reachable from c under h.

For the base case, suppose $n = 1$. Then only c is reachable from c under h. Then since c is in V, $h(c)$ must contain c. Then c is recursive under h, so c is in V'.

So say $n \geq 1$. Suppose for induction that for every c' in V and $n' < n$ such that n' is the number of symbols reachable from c' under h, c' is in V'. Since c is in V, $h(c)$ contains some $d \in V$. Suppose c is reachable from d. Then c is recursive under h, so c is in V'. So say c is not reachable from d. Then since every symbol reachable from d is reachable from c, the number of symbols reachable from d is at most $n-1$. So by the induction hypothesis, d is in V'. Then c is in V', completing the induction.

So $M \subseteq M'$ and $V \subseteq V'$. Then since $M \cup V = M' \cup V' = A$ and $M' \cap V' = \{\}$, $M' = M$ and $V' = V$. □

Theorem 23. *Every infinite 0L system has an infinite D0L subsystem.*

Proof. Take any infinite 0L system $G = (A, \sigma, w)$. By Corollary 4, there is a derivation $D : s_0 \to \cdots \to s_k \to \cdots \to s_n$ such that $0 \leq k < n$, $s_0 = w$, s_k contains a $c \in A$ whose descendant string in s_n contains distinct occurrences of c and a vital symbol d, and no shorter derivation has these properties. The c in s_n has an ancestor symbol c_i in each s_i, and each c_i generates a string x_i contained in s_{i+1}, for $0 \leq i < n$. Similarly, the d in s_n has an ancestor symbol d_i in each s_i, for $0 \leq i < n$. Let m be the highest i such that $k \leq i < n$ and the d in s_n is descended from c_i. Intuitively, m designates the string containing the last common ancestor of the c and d in s_n. Let h be a morphism on A constructed as follows. First, for i from $n-1$ down to 0, set $h(c_i) = x_i$ unless $h(c_i)$ has already been set. Then for all $e \in A$ for which $h(e)$ has not been set, if some $s \in \sigma(e)$ contains a vital symbol, set $h(e)$ to any such s, otherwise set $h(e)$ to any $s \in \sigma(e)$. Then $G' = (A, h, w)$ is a D0L subsystem of G. We will show that G' is infinite.

First we show that c is reachable from w under h. Take any i such that $0 \leq i < n$. We will show by induction that c is reachable from c_i under h. For the base case, suppose $i = n-1$. Clearly c is reachable from c_{n-1} under h, since $h(c_{n-1}) = x_{n-1}$, which contains c. So say $i < n-1$. Suppose for induction that for all j such that $i < j < n$, c is reachable from c_j under h. Suppose there is a j such that $i < j < n$ and $c_j = c_i$. Then by the induction hypothesis, c is reachable from $c_j = c_i$ under h. So say there is no such j. Then $h(c_i) = x_i$, which contains c_{i+1}. By the induction hypothesis, c is reachable from c_{i+1} under h, hence c is reachable from c_i under h, completing the induction. So c is reachable from c_0 under h, hence c is reachable from w under h.

Next we show that there are no i, j such that $k \leq i < j < n$ and $c_i = c_j$. Suppose there are such i, j. Suppose $k \leq i < j \leq m$. Then c_m could be derived from c_i in $m - j$ steps instead of $m - i$ steps, so D is not minimal, a contradiction. So suppose $m < i < j < n$. Then the steps from i to j could be skipped, so that at step $n - (j - i)$, c_i would reach c and d_i would reach $d_{n-(j-i)}$, which is vital. But then D is not minimal. So suppose $k \leq i \leq m < j < n$. Then the descendant string in s_j of the c_i in s_i contains distinct occurrences of c_i and d_j. But then D could be shortened from length n to length j. So there are no such i, j.

So for all $k \leq i < n$, $h(c_i) = x_i$. Hence c_m is reachable from c under h and c is reachable from c_{m+1} under h. Now $h(c_m) = x_m$, which contains distinct occurrences of c_{m+1} and d_{m+1}. By Lemma 22, d_{m+1} is vital under h. Then some string s containing distinct occurrences of c and a vital symbol can be derived from x_m under h. Then some string containing s can be derived from c under h. Hence c is recursive under h, and not monorecursive under h.

Then since c is reachable, recursive, and not monorecursive in G', G' is infinite by Lemma 9. Therefore every infinite 0L system has an infinite D0L subsystem.

□

5.2 DT0L

Let $G = (A, H, w)$ be a DT0L system. A **D0L subsystem** of G is a D0L system $G' = (A, h, w)$ such that h is in H. Notice that $L(G') \subseteq L(G)$.

Theorem 24. *There is an infinite DT0L system with no infinite D0L subsystem.*

Proof. Let $A = \{\mathtt{a}, \mathtt{b}\}$. Let h_1 and h_2 be morphisms on A such that $h_1(\mathtt{a}) = \mathtt{ab}$, $h_1(\mathtt{b}) = \lambda$, $h_2(\mathtt{a}) = \lambda$, and $h_2(\mathtt{b}) = \mathtt{bb}$. Let $H = \{h_1, h_2\}$. Let $w = \mathtt{a}$. Then the DT0L system (A, H, w) is infinite but has no infinite D0L subsystem. □

6 Conclusion

In this paper we have extended to 0L, DT0L, and T0L systems the work of Vitányi [7] on infiniteness and boundedness of D0L systems. In doing so, we relaxed the condition of alphabet size-boundedness (which holds for the class of finite D0L systems) to one of alphabet step-boundedness (which holds also for the classes of finite 0L and DT0L systems). One direction for further work would be to find a related boundedness condition which holds for the class of finite T0L systems. We have also shown that every infinite T0L language has an infinite D0L subset, and that every infinite 0L system has an infinite D0L subsystem. It would be interesting to see whether in classes of L systems beyond the ones studied here, infiniteness is similarly characterized by the presence of notable infinite subsets and subsystems.

Acknowledgments. I want to thank my advisor, Rajmohan Rajaraman, for supporting this work, encouraging me, and offering many helpful comments and suggestions.

References

1. Jungers, R.M., Protasov, V., Blondel, V.D.: Efficient algorithms for deciding the type of growth of products of integer matrices. Linear Algebra and its Applications 428, 2296–2311 (2008)
2. Kari, L., Rozenberg, G., Salomaa, A.: L systems. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 1, pp. 253–328. Springer-Verlag New York, Inc., New York (1997)
3. Nishida, T.: Quasi-Deterministic 0L Systems. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 65–76. Springer, Heidelberg (1992)
4. Nishida, T.Y., Salomaa, A.: Slender 0L languages. Theoretical Computer Science 158(1-2), 161–176 (1996)
5. Rabkin, M.: Ogden's Lemma for ET0L Languages. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 458–467. Springer, Heidelberg (2012)
6. Rozenberg, G., Salomaa, A.: Mathematical Theory of L Systems. Academic Press, Inc., Orlando (1980)
7. Vitányi, P.: On the Size of D0L Languages. In: Rozenberg, G., Salomaa, A. (eds.) L Systems. LNCS, vol. 15, pp. 78–92. Springer, Heidelberg (1974)

Uniformisation of Two-Way Transducers

Rodrigo de Souza[⋆]

DEINFO/EADTec, Universidade Federal Rural de Pernambuco, Recife, Brazil
rsouza@deinfo.ufrpe.br

Abstract. We show that every relation realised by a nondeterministic two-way transducer contains a function with equal domain which can be realised by a sequential two-way transducer. Our proof is built on three structural constructions with automata: a variant of Shepherdson's method to convert a two-way automaton into an equivalent one-way automaton, which we call the folding of a two-way automaton; the construction of an unambiguous automaton for a rational language based on covering of automata; a simulation of an unambiguous automaton by a deterministic two-way one due to Hopcroft and Ullman. It follows a new proof for the fact (due to Engelfriet and Hoogeboom) that every functional two-way transducer can be converted into a sequential one, together with a clear estimation for the underlying complexity.

Keywords: two-way transducer, determinisation, uniformisation.

1 Introduction

How nondeterminism can be more powerful than determinism is a recurrent question in Automata Theory and has been established for plenty of models of computation. Unlike classical automata, *finite-state transducers*, or two-tape automata which realise relations between words, cannot be determinised: a deep result of Choffrut [1] shows that the word-to-word functions which can be realised by a *sequential*[1] transducer form a strict subset of the *rational functions* (= the behaviour of functional, or single-valued, transducers). One of our purposes is to give a new proof for a result of Engelfriet and Hoogeboom (Theorem 22 in [5]) which settles the matter for another importante model, the *two-way transducers*:

Theorem 1 (Engelfriet-Hoogeboom 2001). *Functional two-way transducers can be effectively turned into equivalent sequential two-way transducers.*

A two-way transducer is a two-way automaton with outputs on the transitions, that is: the reading of the input word is a sequence of left or right moves, the resulting output word is the one-way concatenation of the corresponding outputs; a two-way transducer is said to be *sequential* if its underlying input automaton is deterministic. Two-way automata have been introduced in the seminal paper

[⋆] Supported by *Fundação de Amparo à Ciência do Estado de Pernambuco* (FACEPE).

[1] Or *input-deterministic*. As in [10] (see Remark 5), we prefer to call such transducers *sequential*, avoiding the traditional term *subsequential*.

of Rabin and Scott [11] together with the proof that they have the same recognition power of one-way automata; a more accessible proof has been given by Shepherdson [16]. For transducers however the two-way model is more powerful, and one can find very simple functions, such as the mirror function, which can be realised by a two-way transducer but are not rational. Theorem 1 states that the sequential two-way transducers encompass all the functions which can be realised by a (one or two-way) transducer.

Actually, our main result (Theorem 3) is a language-theoretical property of two-way transducers from which Theorem 1 is a direct consequence. It recalls a classical property of transducers, the fact that for every rational relation τ there exists a *rational function* which "chooses" for every word in the domain of τ exactly one word in its image. It is what we call a rational *uniformisation* of τ:

Theorem 2 (Rational Uniformisation Theorem). *Every relation realised by a (one-way) transducer can be uniformised by a functional transducer.*

The first proof of Theorem 2 dates back to 1969 [9] together with one of its remarkable consequences: *every rational function can be realised by an unambiguous transducer* (= distinct successful computations have distinct labels). It is an old result, but the topic is far from being closed: at least two new proofs followed and the property has been investigated within other contexts (trees and infinite words) [2]. In our main result, we lift the Rational Uniformisation Theorem to the *two-way transducers* with a more precise statement:

Theorem 3. *Every relation realised by a two-way transducer can be uniformised by a function realised by a* sequential *two-way transducer.*

Our proof of Theorem 3 bears some similarities to Elgot and Mezei's classical representation of a rational function as the composition of a left and a right sequential ones [4]. Starting from a two-way transducer $\mathcal{T}$, we define two sequential two-way transducers, a *left* and a *right pathfinder* in our terminology (Sect. 4). Left pathfinders have been implicitly defined by Hopcroft and Ullman to prove that languages realised by certain automata are closed under inverse deterministic gsm mappings (Theorem 5 in [7]). Here, they are used to find, for every input word u, the sequence of moves of a successful computation of $\mathcal{T}$ reading u. At each step, if the next move is a left one, the transition is discovered by the right pathfinder, and vice-versa. This task is an involved routine: the pathfinder scans a specific part of u and then has to come back to the original position.

But before we need to select, for every input word, *the* successful computation, among all those reading the word, which is going to be simulated. To this end, we use two distinct constructions to define an unambiguous one-way automaton $\mathcal{B}$ having exactly one successful computation for every word in the domain of $\mathcal{T}$ (Theorem 4). The first construction is a variant of Shepherdson's method to turn a two-way automaton into an equivalent *one-way* automaton, which we call the *folding* of a two-way automaton. Applied to the underlying (two-way) input automaton of $\mathcal{T}$, it yields a one-way automaton $\mathcal{S}$, whose successful computations are a kind of folding of the computations of $\mathcal{T}$. Next, we use in $\mathcal{S}$ the *lexicographic covering* construction we defined in [15] to decompose a k-valued transducer into a union of k functional ones. It yields a new automaton which "covers" $\mathcal{S}$, in the sense that its successful computations project bijectively into those of $\mathcal{S}$ – this is what we call a covering of automata. Then, we can extract from the covering the unambiguous $\mathcal{B}$ to be used in pathfinder's construction (Sect. 3). The whole process is summarised in Fig. 1. The proof of Theorem 1 and an application to the *equivalence problem* are discussed in Sect. 6.

$$\begin{tikzcd}
 & & & & \mathcal{U}=\mathcal{T}\times\mathcal{L}\times\mathcal{R} \arrow[dl] \arrow[dr] \arrow[ddllll] & \\
 & & & \mathcal{L} \arrow[dr] & & \mathcal{R} \arrow[dl] \\
\mathcal{T} & \mathcal{A} \arrow[l] & \mathcal{S} \arrow[l] & & \mathcal{B} \arrow[ll] &
\end{tikzcd}$$

Fig. 1. Construction of a sequential two-way transducer $\mathcal{U}$ which uniformises the two-way transducer $\mathcal{T}$: $\mathcal{A}$ is the underlying input automaton of $\mathcal{T}$; $\mathcal{S}$ is the folding of $\mathcal{A}$; $\mathcal{B}$ is an unambiguous automaton given by a lexicographic covering of $\mathcal{S}$; $\mathcal{L}$ and $\mathcal{R}$ are respectively a left and a right pathfinder built over $\mathcal{B}$; $\mathcal{U}$ is a product of $\mathcal{T}$, $\mathcal{L}$ and $\mathcal{R}$.

The original proof of Theorem 1 in [5] appears within a logical framework developed for string and graph transducers; it is based on operations between certain monadic second-order logic definable graph relations. The main feature of our proof is that it is built on constructions which depend directly on the *structure* of the involved transducers. It follows a clear estimation for the complexity of the resulting algorithm (whereas no bound is given in [5]): four exponentials on the size of $\mathcal{T}$. Indeed, foldings and lexicographic coverings have exponential size, the pathfinder construction provokes two exponentials. We hope that a better understanding of these constructions will provide a more precise estimation.

2 Automata and Transducers, One and Two-Way

As far as classical (one-way) automata and transducers are concerned, we adopt the notation[2] in [14]; for two-way machines, our terminology is close to [8].

One-way automata, or simply automata, are acceptors which read the input tape from left to right; in *one-way transducers* the reading is made in two tapes, also from left to right, and pairs of words are accepted. Both consist of a finite set Q of *states*, sets I, $T \subseteq Q$ of *initial* and *final* states, respectively, and the set E of transitions. In an automaton over the alphabet A, $E \subseteq Q \times A \times Q$, that is, transitions are labelled by letters; in a transducer over the alphabets A and B, the labels are pairs in $A \times B^*$: a letter in the first tape and a word in the second.[3]

[2] In particular, the set of words over a finite alphabet A (the free monoid over A) is denoted by A^* and the empty word by 1.

[3] In general, transducers are labelled by pairs of words, elements of the product monoid $A^* \times B^*$; but for the *finitely-valued* transducers we are dealing with, it is not restrictive to impose that the first component is always a letter [14]. Such a transducer is called a *nondeterministic generalised sequential machine* in some references.

The *behaviour* of an automaton $\mathcal{A}$ is the subset of A^* consisting of the labels of the *successful computations* of[4] $\mathcal{A}$. A computation represents the reading of a word: it is a sequence of consecutive transitions, $c : p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \ldots \xrightarrow{a_\ell} p_\ell$; its label is $a_1 \ldots a_\ell$, and c is *successful* if $p_0 \in I$ and $p_\ell \in T$. For transducers, the label of a computation is the pair obtained by the componentwise concatenation of the labels of the transitions, and the behaviour is a subset of $A^* \times B^*$.

From a "dynamic" point of view, the behaviour of a transducer is a relation from A^* to B^* which sends every word $u \in A^*$ to the set of words $x \in B^*$ such that (u, x) is the label of some successful computation — the *image* of u. In this setting, we say that a computation labelled by (u, x) *reads* u and *writes* x, and that u is its *input* and x its *output*. The transducer is called k*-valued*, where k is a positive integer, if the image of every input word has at most k words. It is *functional* if $k = 1$. Transducers realise what we call *rational relations*: those whose graph is a rational subset of $A^* \times B^*$. See an example in Fig. 2(a).

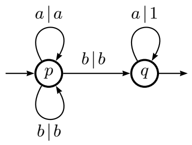
Two-way automata are the same as automata, except that the transitions are labelled by letters "decorated" by a left or a right arrow, and the input word is surrounded by endmarkers: a left endmarker, $\rhd$, and a right one, $\lhd$, which do not belong to the alphabet A (thus, transitions are taken from the set $Q \times (A \cup \{\rhd, \lhd\}) \times \{\leftarrow, \rightarrow\} \times Q$). At each step of the reading the tape head can move left as well as right (we do not permit stationary moves), according to the arrow of the corresponding transition. A successful computation starts at the left endmarker, wanders around the word, possibly visiting the endmarkers, and eventually walks off the right edge of the tape, ending in a final state. Whenever the left endmarker is visited, the automaton makes a right move, and moves left when scanning the right endmarker (except for the last move). A word $u \in A^*$ is accepted by the automaton if there is a successful computation labelled by $\rhd u \lhd$.

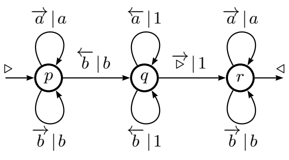
Formally, a computation in a two-way automaton is a sequence of words in $\rhd A^* Q A^* \lhd$, which intend to record the current configuration of the automaton. A configuration xqy means that xy is the input string (including the endmarkers), q is the current state and the input head is scanning the first letter of y (or has moved off the right edge if $y = 1$). A pair of consecutive configurations represent a move of the input head: a right move from $xpay$ $(a \in A)$ to $xaqy$ if the automaton has a transition from p to q labelled by $\overrightarrow{a}$, a left move from $xpay$ to $zqbay$ if $x = zb$ $(b \in A)$ and there is a transition from p to q labelled by $\overleftarrow{a}$.

In a *two-way transducer*, the reading of the first tape has the same rules of a two-way automaton, but the output tape is *one-way*: at each move a word is concatenated at the right of the output written so far. Thus, a two-way transducer is simply a two-way automaton with an output word in the transitions. An example is depicted in Fig. 2(b), together with two computations drawn as a zigzag walk in the plane. In this drawing, configurations where $\mathcal{T}$ is reading the same position in the input word are column-aligned. A two-way transducer is *sequential* if its *underlying input automaton* is deterministic.

A classical construction due to Shepherdson shows that the behaviour of a two-way automaton is rational [16]. But for transducers the two-way ones are

[4] A *rational subset* of A^*.


(a) $\mathcal{F}$


(b) $\mathcal{T}$

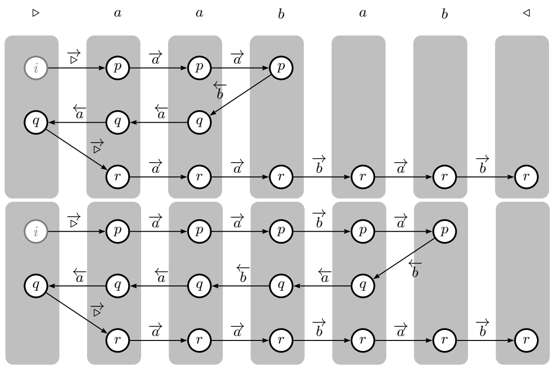

(c)

Fig. 2. In 2(a), a functional and unambiguous transducer, $\mathcal{F}$, which reads words of form uba^n and writes ub. In 2(b), a two-way transducer, $\mathcal{T}$, which sends every word u having at least one b to the set $\{xbu \mid xb \text{ is prefix of } u\}$. In 2(c), two successful computations of $\mathcal{T}$ labelled by $\triangleright aabab \triangleleft$, depicted in such a way that every configuration xsy is represented solely by the state s in the column below the first letter of y (gray rectangle). Initial states are represented by a small ingoing edge, and final states with an outgoing edge. A transition in a transducer labelled by a pair (a, x) is denoted by $a|x$. In $\mathcal{T}$, we do not exhibit the unique initial state i and the final state t; they are implicitly represented by the ingoing arrow in p and the outgoing one in r, respectively.

more powerful, by far: the image of a rational set by a two-way transducer may not even be context-free [12] (for one-way tranducers it is always rational).

A *loop* in a computation c labelled by $u = a_0a_1a_2 \ldots a_\ell a_{\ell+1}$ ($a_0 = \triangleright$, $a_{\ell+1} = \triangleleft$, $a_j \in A$ for $1 \leq j \leq \ell$) is a segment of c which starts and ends at the same configuration; it represents a sequence of moves which starts and comes back to the same position of u. The fact that two-way automata can exhibit erratic behaviour and loop indefinitely will not be an issue for the suppression of a loop yields a legitimate new successful computation with the same label. Thus, every word accepted by the automaton is the label of some loop-free successful computation. For transducers, the suppression of a loop yields a computation with the same input and possibly a shorter output; but in the uniformisation construction we are going to present the outputs play no role. From now on the term "computation" is a shorthand for "loop-free successful computation".

3 From Two-Way to Unambiguous One-Way

The first step of our proof of Theorem 3 is the construction of the *one-way automaton* $\mathcal{S}$ and from it the equivalent *one-way and unambiguous* automaton $\mathcal{B}$ in Fig. 1. The two-way transducer $\mathcal{T}$ and $\mathcal{S}$ are related by means of an operation between computations which we call the *folding of a computation*; we explain it below. The unambiguous $\mathcal{B}$ is obtained from $\mathcal{S}$ via the lexicographic covering construction we defined in [15], and its computations project injectively in those of $\mathcal{S}$. The constructions together provide a proof for the following:

Theorem 4. *For every two-way automaton $\mathcal{A}$, there exists an equivalent and unambiguous one-way automaton $\mathcal{B}$ such that every successful computation of $\mathcal{B}$ is the folding of a (successful and loop-free) computation of $\mathcal{A}$.*

Intuitively, every computation of $\mathcal{S}$ is a folding of a zigzag walking of $\mathcal{A}$ into a straight sequence of column vectors of states. These vectors represent, for every position j of the input word, the states of the walking where the head is scanning j. This is depicted in Fig. 2(c), where a gray rectangle, read from top to bottom, is a tuple of states where $\mathcal{A}$ is scanning the same position.[5]

Let us explain this folding operation formally before proceed to the definition of $\mathcal{S}$. Let c be a computation of $\mathcal{A}$ labelled by $u = a_0a_1a_2 \ldots a_\ell a_{\ell+1}$ ($a_0 = \triangleright$, $a_{\ell+1} = \triangleleft$, $a_j \in A$ for $1 \leq j \leq \ell$). First, we need to define for every position j the *column of c at j*, or simply j-column: let $xq_1y, xq_2y, \ldots, xq_hy$ be the subsequence of configurations of c such that $x = a_0 \ldots a_{j-1}$ (that is, the head is at position j), and $d_1, d_2, \ldots, d_h \in \{\leftarrow, \rightarrow\}$ be the respective directions of the next moves (that is, $d_i = \leftarrow$ if, in c, xq_iy goes to a configuration of form $a_0 \ldots a_{j-2}pa_{j-1} \ldots a_{\ell+1}$, $d_i = \rightarrow$ otherwise); the j-column is the tuple $((q_1, d_1), (q_2, d_2), \ldots, (q_h, d_h)), a_j)$. The reason why the letter a_j is also recorded will be explained below, and the arrows indicating the direction of the next moves will be useful in the pathfinder construction (next section). Notice that, as c is loop-free, the states $q_1, \ldots, q_h$ are pairwise distinct. Now, the folding of c is the sequence of columns at positions $0, 1, 2, \ldots, \ell + 1$, and we can state the main property of the automaton $\mathcal{S}$:

Proposition 5. *The folding operation puts a one–to–one correspondence between the (successful and loop-free) computations of $\mathcal{A}$ and the successful computations of $\mathcal{S}$. That is: every successful computation of $\mathcal{S}$ is the folding of some computation of $\mathcal{A}$; for every computation of $\mathcal{A}$, its folding is a computation of $\mathcal{S}$; the folding operation is injective.*

Let us finally define $\mathcal{S}$. The states are the set of all possible columns. A state $Q = ((q_1d_1), \ldots, (q_h, d_h), a)$ is initial if: q_1 is initial; $d_1, \ldots, d_h$ are all equal to the right arrow; $a = \triangleright$. It is final if $d_1, \ldots, d_{h-1}$ are all equal to the left arrow, $d_h = \rightarrow$ and there is a transition from q_h to a final state labelled by $\overrightarrow{\triangleleft}$ in $\mathcal{S}$. The transitions are the more elaborate part, for a transition from Q to the column $P = ((p_1, e_1), \ldots, (p_k, e_k), b)$ represents the relation between the states $q_1, \ldots, q_h$ and $p_1, \ldots, p_k$ in a computation of $\mathcal{A}$. First of all, all the transitions which start at Q are labelled by a (the letter recorded in the column). Next, let α be the number of forward states in Q, that is, the pairs (q_i, d_i) such that $d_i = \rightarrow$, and β be the number of backward states in P ($e_i = \leftarrow$). There is a transition from Q to P if the following hold: $d_1 = \rightarrow$ and there is a transition from q_1 to p_1 labelled

[5] There is a clear resemblance between these tuples and the concept of *crossing sequence* in Shepherdson's two-way to one-way construction [16]. Actually the original concept is slightly different. A crossing sequence is formed by states which "crosses" the "boundary" between two consecutive positions. For more details see [8].

by $\overrightarrow{a}$ in $\mathcal{A}$; for every k, $2 \leq k \leq \alpha$, there is a transition labelled by $\overrightarrow{a}$ from the k-th forward state of Q to the state which follows the $(k-1)$-th backward state of P; for every ℓ, $1 \leq \ell \leq \beta$, there is a transition labelled by $\overleftarrow{b}$ from the ℓ-th backward state of P to the state which follows the ℓ-th forward state in Q.

Let us sketch the proof that the computations of $\mathcal{S}$ are precisely the foldings of the computations of $\mathcal{A}$ — Proposition 5. That the folding of a computation of $\mathcal{A}$ forms a computation of $\mathcal{S}$ comes from the analysis of consecutive columns in the folding: the moves between the corresponding configurations match exactly the conditions which define the transitions of $\mathcal{S}$. Next, one can show that every computation of $\mathcal{S}$ can be "unfolded" into a computation of $\mathcal{A}$. This unfolding operation yields a graph where the vertices are the several configurations associated with the columns, and the arcs are given by the conditions which define the transitions of $\mathcal{S}$. Such a graph is precisely a computation of $\mathcal{A}$.

We obtain the unambiguous automaton $\mathcal{B}$ with the construction of a *covering* over $\mathcal{S}$. Covering of automata have been introduced in [13] and since then are used systematically to tackle several problems in automata and transducers. Intuitively, starting from an automaton $\mathcal{A}$, an *expansion* of $\mathcal{A}$ is performed and yields a larger automaton $\mathcal{C}$, which is a *covering* of $\mathcal{A}$; this means that the computations of $\mathcal{C}$ and those of $\mathcal{A}$ are in a $1-1$ correspondence and that they have, roughly speaking, the same "structure". In the larger $\mathcal{C}$, it is so to say easier to distinguish between the computations and the proof of the property aimed at by the construction amounts to an adequate choice within these computations. The formal definition can be seen in [14]. To construct the unambiguous $\mathcal{B}$ which establishes Theorem 4, the lexicographic covering we used in [15] to decompose bounded-valued transducers come at hand. This is illustrated in Fig. 3.

4 From Unambiguous One-Way to Sequential Two-Way

Every nondeterministic automaton $\mathcal{A}$ can be converted into a deterministic one $\mathcal{A}_{\mathsf{det}}$ by means of the subset construction.[6] But let us suppose that $\mathcal{A}$ is unambiguous (no two distinct successful computations with the same label) and besides the deterministic reading of the input word, we want to know precisely *the* successful computation of $\mathcal{A}$ labelled by this word. Hopcroft and Ullman's proof that languages realised by two-way balloon automata are closed by inverse gsm mappings [7] amounts to the construction of a two-way automaton which performs such a deterministic reading, and fits perfectly to the deterministic simulation of computations we need to establish Theorem 3. This is what we call a *pathfinder automaton*: a sequential two-way automaton which finds successively the transitions of the successful computation labelled by the input word.

Roughly speaking, when scanning the position j of the word $\rhd a_1 \ldots a_\ell \lhd$ (where every a_j is a letter of A, the alphabet of $\mathcal{A}$), the configuration of the pathfinder stores the j-th state of the unique successful computation of $\mathcal{A}$ labelled by $a_1 \ldots a_\ell$. Then, the pathfinder performs a round of two sequence of moves: first,

[6] In our notation, $\mathcal{A}_{\mathsf{det}}$ is the *accessible part* of the subset construction.

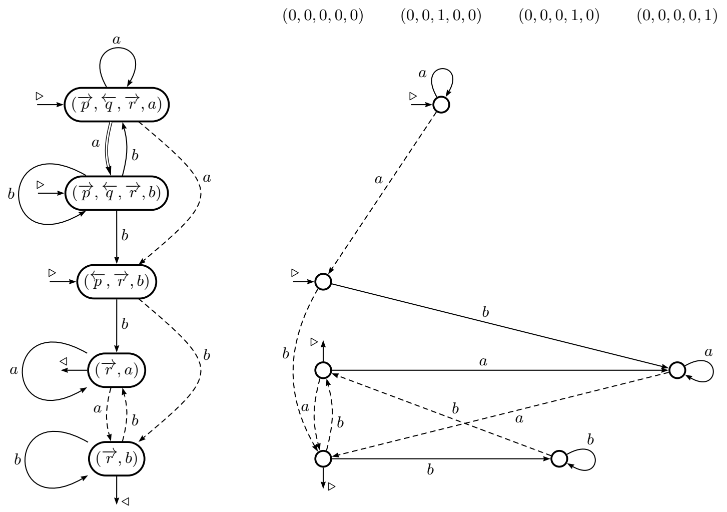

(0, 0, 0, 0, 0)
(0, 0, 1, 0, 0)
(0, 0, 0, 1, 0)
(0, 0, 0, 0, 1)
$(\overrightarrow{p}, \overleftarrow{q}, \overrightarrow{r}, a)$
$(\overrightarrow{p}, \overleftarrow{q}, \overrightarrow{r}, b)$
$(\overleftarrow{p}, \overrightarrow{r}, b)$
$(\overrightarrow{r}, a)$
$(\overrightarrow{r}, b)$
a
b

Fig. 3. At left, the folding $\mathcal{S}$ of the underlying input automaton of the two-way transducer $\mathcal{T}$ in Fig. 2(b) (the initial columns are not explicitly drawn, but are represented by the ingoing edges labelled by $\triangleright$); at right, the unambiguous automaton extracted from a lexicographic covering of $\mathcal{S}$. See [15] for a detailed explanation of the covering construction. The ordering of the transitions is: dashed < solid < double.

it moves the head forward further enough – possibly the end of the word – until it finds a configuration where it can be "seen" the way to the next state of the computation; then, it comes back to the position j. But the pathfinder cannot store this position: it has to find it again. It does it by means of a trick involving the computations of the automaton $\mathcal{A}^{\varrho}_{\mathsf{det}}$ obtained by *reversing* $\mathcal{A}$ *and applying the subset construction.* We shall explain this construction in a context close to the matter of this paper by describing a proof for the following:

Theorem 6. *Sequential two-way transducers realise all the rational functions.*

That is, for every unambiguous and one-way functional transducer[7] $\mathcal{F}$ one can build an equivalent sequential two-way transducer. The pathfinder for $\mathcal{F}$ does this task: for every input word u, it finds successively the transitions of the unique successful computation of $\mathcal{F}$ reading u and writes their outputs.

We need some notation. Let $\mathcal{A}$ be the underlying input automaton of $\mathcal{F}$. For every deterministic automaton used in this text, we denote by a dot the extended transition function (which is a right action of A^* over the states of the automaton). E.g. the state reached from the initial state I of $\mathcal{A}_{\mathsf{det}}$ with the reading of the word x is $I \cdot x$. Symmetrically, see $\mathcal{A}^{\varrho}_{\mathsf{det}}$ as an automaton which reads from right to left[8] starting at its initial state J, and for every $y \in A^*$ denote by $y \cdot J$ the state of $\mathcal{A}^{\varrho}_{\mathsf{det}}$ reached with the reading of the reversal of y.

[7] Recall that the rational functions are unambiguous [14].

[8] Although $\mathcal{A}^{\varrho}_{\mathsf{det}}$ is a legitimate one-way automaton, which recognises the reverse of the words accepted by $\mathcal{A}$.

For readability, we split the states of a pathfinder into three sets, which we call *modes*: B, the *booting mode*, or B-mode; S, the *scanning mode*, or S-mode; D, the *discovering mode*, or D-mode (we also say D-states, etc.). Let us at first describe the S-mode, the starting of every round of moves allowing to find the next state of the computation of $\mathcal{A}$ being simulated. It consists simply of states of the Cartesian product of automata $\mathcal{A}_{\mathsf{det}} \times \mathcal{A}^{\varrho}_{\mathsf{det}}$. Its relevant property is an invariant: whenever the pathfinder is in an S-state (R, S) and scanning the position j of the input word $u = a_1 \ldots a_\ell$, R is the state $I \cdot (a_1 \ldots a_{j-1})$ and S the state $(a_j \ldots a_\ell) \cdot J$. It follows that $R \cap S$ *contains an unique state*: indeed, $I \cdot (a_1 \ldots a_{j-1})$ is the set of states that $\mathcal{A}$ can reach from some initial state with a computation reading $a_1 \ldots a_{j-1}$; $(a_j \ldots a_\ell) \cdot J$ is the set of states which can reach some final state with a computation reading $a_j \ldots a_\ell$; distinct states in $R \cap S$ would then imply distinct successful computations reading u; but $\mathcal{A}$ is unambiguous. The unique state p in $R \cap S$ is precisely the j-th state of the computation of $\mathcal{A}$ reading u. For simplicity, we denote $I \cdot (a_1 \ldots a_{j-1})$ by $I\langle j-1 \rangle$ and $(a_j \ldots a_\ell) \cdot J$ by $\langle j \rangle J$. The computation of the pathfinder starts at the B-mode, which simply reads the entire u from left to right, and next from right to left simulating $\mathcal{A}^{\varrho}_{\mathsf{det}}$ (starting at the initial J), to find the S-state $(I, \langle 1 \rangle J)$.

The crucial task of the pathfinder is to pass from the S-state $(I\langle j-1 \rangle, \langle j \rangle J)$ to $(I\langle j \rangle, \langle j+1 \rangle J)$, for every position j. In the first component, it simply follows the transition in the deterministic $\mathcal{A}_{\mathsf{det}}$ labelled by a_j, that is, $I\langle j \rangle$ is $I\langle j-1 \rangle \cdot a_j$. In the second component, it may happen that there is exactly one transition in $\mathcal{A}^{\varrho}_{\mathsf{det}}$ of form $X \xrightarrow{a_j} \langle j \rangle J$. In this easy case, X is $\langle j+1 \rangle J$ and the next state of the computation of $\mathcal{F}$ reading u is the unique element q in $(I\langle j-1 \rangle \cdot a_j) \cap X$. The pathfinder goes to the new S-state $(I\langle j-1 \rangle \cdot a_j, X)$, moves the head to the right and writes the output of the transition of $\mathcal{F}$ from p to q reading a_j.

But it may happen as well that there exists in $\mathcal{A}^{\varrho}_{\mathsf{det}}$ two or more transitions $X_1 \xrightarrow{a_j} \langle j \rangle J, \ldots, X_t \xrightarrow{a_j} \langle j \rangle J$. Here, exactly one of the X_i's is $\langle j+1 \rangle J$. In order to discover it, the pathfinder enters the D-mode. There are two kinds of D-states, the forward states and the backwards ones; the pathfinder at first uses the former to perform a sequence of forward moves, and next the latter for a sequence of backward moves. Let us at first explain the moves of the forward states. They are based on the automaton $\mathcal{C}$ obtained by *reversing* $\mathcal{A}^{\varrho}_{\mathsf{det}}$ *again and applying the subset construction.*[9] For every X_i, $1 \leq i \leq t$, denote by $\mathcal{C}(X_i)$ the part of $\mathcal{C}$ accessible from X_i; the forward D-states are the states of the product of automata $\mathcal{C}(X_1) \times \ldots \times \mathcal{C}(X_t)$ (actually, they also store the current S-state $(I\langle j-1 \rangle, \langle j \rangle J)$, but we do not write it explicitly to avoid a clumsier notation). Starting from the forward D-state $X_1 \times \ldots X_t$ and the position j, the pathfinder simulates the deterministic[10] $\mathcal{C}(X_1) \times \ldots \times \mathcal{C}(X_t)$, that is: it scans successively the positions $j+1, j+2, \ldots$ of the input word u and visits the states $(X_1 \cdot a_{j+1}) \times \ldots \times (X_t \cdot a_{j+1}), (X_1 \cdot a_{j+1}a_{j+2}) \times \ldots \times (X_t \cdot a_{j+1}a_{j+2}), \ldots$. Here we can state the main property which allows to discover $\langle j+1 \rangle J$: either *at*

[9] Let us remark that what we have done for $\mathcal{C}$ is Brozozowski's double reversal construction of the minimal automaton for a rational language.

[10] The product of deterministic automata is also a deterministic automaton.

some position $k > j$ exactly one of the sets $(X_i \cdot a_{j+1} \dots a_k)$ is nonempty, or *the reading reaches $(X_1 \cdot a_{j+1} \dots a_\ell) \times \dots \times (X_t \cdot a_{j+1} \dots a_\ell)$ (at the end of u), and J, the initial state of $\mathcal{A}^{\varrho}_{\mathsf{det}}$, is contained in exactly one set $X_i \cdot a_{j+1} \dots a_\ell$.* This fact is a consequence of the unambiguity of $\mathcal{A}^{\varrho}_{\mathsf{det}}$. We call such a set the target set; *its states are precisely the ones which reach $\langle j+1 \rangle J$ after the right to left reading from the current position to j in the automaton $\mathcal{A}^{\varrho}_{\mathsf{det}}$*. In either case, the pathfinder goes to a backward D-state. These backwards states belong to the product $\mathcal{A}^{\varrho}_{\mathsf{det}} \times \mathcal{A}^{\varrho}_{\mathsf{det}}$, and every pair (Y, Z) in the deterministic backward reading is such that: Y comes from an arbitrary fixed state belonging to the target set; Z comes from an arbitrary fixed state belonging to some other set (among the ones reached by the forward moves). *We have that $Y \neq Z$ for every position greater than j, and $Y = Z$ in position j* — this allows to find the position j again. In this position, $\langle j+1 \rangle J$ is known — it is the state reached from the target set — and the pathfinder writes the output of the transition of $\mathcal{F}$ from the unique state in $I\langle j-1 \rangle \cap \langle j \rangle J$ to the unique one in $I\langle j \rangle \cap \langle j+1 \rangle J$. The D-mode ends, the pathfinder moves the head to the right (position $j+1$) and enters again the S-mode, with state $(I\langle j \rangle, \langle j+1 \rangle J)$. An example is depicted in Fig. 4.

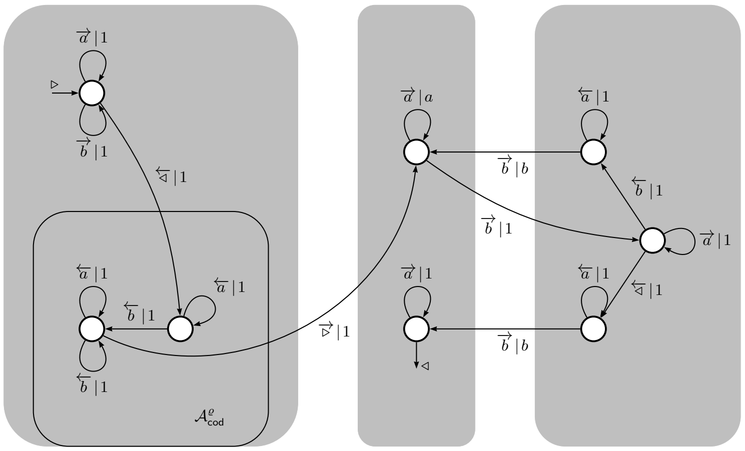

$\overrightarrow{a}\,|\,1$
$\triangleright$
$\overrightarrow{b}\,|\,1$
$\overleftarrow{\triangleleft}\,|\,1$
$\overleftarrow{a}\,|\,1$
$\overleftarrow{a}\,|\,1$
$\overleftarrow{b}\,|\,1$
$\overleftarrow{b}\,|\,1$
$\mathcal{A}^{\varrho}_{\mathsf{cod}}$
$\overrightarrow{\triangleright}\,|\,1$
$\overrightarrow{a}\,|\,a$
$\overrightarrow{b}\,|\,b$
$\overrightarrow{b}\,|\,1$
$\overrightarrow{a}\,|\,1$
$\triangleleft$
$\overrightarrow{b}\,|\,b$
$\overleftarrow{a}\,|\,1$
$\overleftarrow{b}\,|\,1$
$\overrightarrow{a}\,|\,1$
$\overleftarrow{a}\,|\,1$
$\overleftarrow{\triangleleft}\,|\,1$

Fig. 4. The (left) pathfinder $\mathcal{L}$ for the functional transducer $\mathcal{F}$ depicted in Fig. 2(a). The gray rectangles show the states of the three modes of $\mathcal{L}$. Note that the underlying input automaton $\mathcal{A}$ of $\mathcal{F}$ is co-deterministic, so $\mathcal{A} = \mathcal{A}_{\mathsf{cod}}$.

The pathfinder we have just explained is a left-to-right one. In the same manner we can define its dual, the *right pathfinder*, which for every computation $p_0 \xrightarrow{a_1|x_1} p_1 \xrightarrow{a_2|x_2} \dots \xrightarrow{a_n|x_n} p_n$ of $\mathcal{F}$ starts the reading at right, and finds successively the transitions labelled by a_n, a_{n-1}, etc. It is the combination of a left and a right pathfinder which will allow the bidirectionality needed to find successful computations in a two-way nondeterministic transducer.

5 Constructing the Uniformisation

Now we explain how the constructions we have just presented can be combined in order to construct a sequential two-way transducer which realises the uniformisation of a two-way transducer — Theorem 3. Let us recall the involved automata (see Fig. 1): $\mathcal{T}$ is the original nondeterministic two-way transducer; $\mathcal{L}$ and $\mathcal{R}$ are respectively the left and right pathfinders for the unambiguous automaton $\mathcal{B}$. The uniformisation is realised by the sequential two-way transducer $\mathcal{U}$, which is a kind of product of $\mathcal{T}$, $\mathcal{L}$ and $\mathcal{R}$. Intuitively, during the reading of the input word $u = a_1 \dots a_\ell$, $\mathcal{U}$ performs the moves represented by the unique computation of $\mathcal{B}$ labelled by u, which is the folding of a computation of $\mathcal{T}$. It "calls" the discovering mode of $\mathcal{L}$ and $\mathcal{R}$ alternatively, depending on the direction indicated by the states of $\mathcal{T}$ in the columns which form this computation.

The transducer $\mathcal{U}$ has also an S-mode, which is formed from the states of $\mathcal{B}_{\mathsf{det}}$ (the subset construction applied to $\mathcal{B}$) and $\mathcal{B}^{\varrho}_{\mathsf{det}}$ together with an state of $\mathcal{T}$. For every position j of u, if $\mathcal{U}$ is in the S-state $(I\langle j-1\rangle, \langle j\rangle J, p)$, then $I\langle j-1\rangle \cap \langle j\rangle J$ has exactly one element, *the j-th column of the computation of $\mathcal{B}$ labelled by u*, and p is an state of this column.[11] The arrow attached to p points the direction of the next move. If it is a right arrow, $\mathcal{U}$ enters the D-mode of $\mathcal{L}$ to find the next S-state $(I\langle j\rangle, \langle j+1\rangle J, q)$, where q is the end of some transition of $\mathcal{T}$ from p to q with input $\overrightarrow{a_j}$; the pathfinder goes to position $j+1$ and writes the output of this transition. Symmetrically, if the arrow is left, $\mathcal{U}$ enters the D-mode of $\mathcal{R}$, finds the previous S-state $(I\langle j-2\rangle, \langle j-1\rangle J, q)$ and goes to the position $j-1$.

6 Applications

Theorem 3 implies that for two-way transducers nondeterminism is not more powerful than sequentiality — Theorem 1. Indeed, our construction applied to a functional two-way transducer yields an *equivalent* sequential transducer, for *the uniformisation of a function is exactly the same function.*

We also note that our proof for Theorem 1 points to a new, more structural proof for the *equivalence problem for functional two-way transducers*:

Theorem 7 (Culik–Karhumäki 1987 [3]). *The equivalence problem for nondeterministic functional two-way transducers is decidable.*

The realm of Culik and Karhumäki's proof is language theory: it is based on a compactness property of systems of word equations known as Ehrenfeucht Conjecture. Our machinery helps in reducing the problem to a (hopefully) simpler one: the equivalence for sequential two-way transducers, which is known to be decidable as shown by Gurari with complexity theory arguments [6].

Acknowledgements. We thank one of the referees for the careful reading of the manuscript and for bringing the work of Engelfriet and Hoogeboom to our attention.

[11] To be more precise, the states of $\mathcal{B}$ are not columns, but each one projects to some column of $\mathcal{S}$ — this is a property of the lexicographic covering.

References

1. Choffrut, C.: Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles. Theoretical Computer Science 5(3), 325–337 (1977), `http://dx.doi.org/10.1016/0304-39757790049-4`
2. Choffrut, C., Grigorieff, S.: Uniformization of rational relations. In: Jewels are Forever, pp. 59–71. Springer (1999)
3. Culik, K., Karhumaki, J.: The Equivalence Problem for Single-Valued Two-Way Transducers (on NPDTOL Languages) is Decidable. SIAM Journal on Computing 16(2), 221–230 (1987), `http://dx.doi.org/10.1137/0216018`
4. Elgot, C.C., Mezei, J.E.: On Relations Defined by Generalized Finite Automata. IBM Journal of Research and Development 9(1), 47–68 (1965)
5. Engelfriet, J., Hoogeboom, H.J.: MSO definable string transductions and two-way finite-state transducers. ACM Trans. Comput. Logic 2(2), 216–254 (2001), `http://dx.doi.org/10.1145/371316.371512`
6. Gurari, E.M.: The Equivalence Problem for Deterministic Two-Way Sequential Transducers is Decidable. SIAM Journal on Computing 11(3), 448–452 (1982), `http://dx.doi.org/10.1137/0211035`
7. Hopcroft, J.E., Ullman, J.D.: An approach to a unified theory of automata. In: 8th Annual Symposium on Switching and Automata Theory (SWAT 1967), pp. 140–147. IEEE Computer Society (1967)
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979), `http://www.worldcat.org/oclc/4549363`
9. Kobayashi, K.: Classification of formal languages by functional binary transductions. Information and Control 15(1), 95–109 (1969), `http://dx.doi.org/10.1016/S0019-99586990651-2`
10. Lombardy, S., Sakarovitch, J.: Sequential? Theoretical Computer Science 356(1-2), 224–244 (2006), `http://dx.doi.org/10.1016/j.tcs.2006.01.028`
11. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM J. Res. Dev. 3(2), 114–125 (1959), `http://dx.doi.org/10.1147/rd.32.0114`
12. Rajlich, V.: Absolutely parallel grammars and two-way finite-state transducers. Journal of Computer and System Sciences 6(4), 324–342 (1972), `http://dx.doi.org/10.1016/S0022-00007280025-4`
13. Sakarovitch, J.: A construction on finite automata that has remained hidden. Theoretical Computer Science 204(1-2), 205–231 (1998), `http://dx.doi.org/10.1016/S0304-39759800040-1`
14. Sakarovitch, J.: Elements of Automata Theory. Cambdrige University Press (2009)
15. Sakarovitch, J., de Souza, R.: Lexicographic decomposition of k-valued transducers. Theory of Computing Systems 47(3), 758–785 (2010), `http://dx.doi.org/10.1007/s00224-009-9206-6`
16. Shepherdson, J.C.: The reduction of two-way automata to one-way automata. IBM J. Res. Dev. 3, 198–200 (1959), `http://dx.doi.org/10.1147/rd.32.0198`

A Conditional Superpolynomial Lower Bound for Extended Resolution

Olga Tveretina

School of Computer Science, University of Hertfordshire, UK
o.tveretina@herts.ac.uk

Abstract. Extended resolution is a propositional proof system that simulates polynomially very powerful proof systems such as Frege systems and extended Frege systems. Feasible interpolation has been one of the most promising approaches for proving lower bounds for propositional proof systems and for bounded arithmetic. We show that an extended resolution refutation of an unsatisfiable CNF representing the clique-graph colouring principle admits feasible interpolation. It gives us a conditional result: If there is a superpolynomial lower bound on the non-monotone circuit size for this class of formulas then extended resolution has a superpolynomial lower bound.

1 Introduction

One of the most important questions in propositional proof complexity is to show that there is a family of propositional tautologies requiring superpolynomial size proofs in a strong proof system such as a Frege or extended Frege proof system. It is an open problem to understand which methods can be used to prove lower bounds for these systems.

In recent years the feasible interpolation method had been one of the promising approaches for proving lower bounds for propositional proof systems. Krajícek defined in [9] the connection between proof systems having feasible interpolation and the circuit complexity. The idea behind the methods is as follows. We say that a proof system P admits feasible interpolation if whenever P has a polynomial size refutation of a formula φ, an interpolation function associated with φ has a polynomial size circuit. It is still unknown whether this approach works for strong propositional proof systems such as Frege systems.

There are conditional results stating that feasible interpolation cannot be used for proving lower bounds for powerful proof systems. Krajícek and Pudlák considered formulas based on the RSA cryptographic scheme and showed that unless RSA is not secure, extended Frege systems do not have the feasible interpolation property. Bonet, Pitassi and Raz showed that unless factoring is computable by polynomial-size circuits neither Frege nor TC^0 Frege systems admit feasible interpolation [2]. Bonet et al. considered formulas based on the Diffie-Hellman secret key exchange scheme [8] and proved that if TC^0 Frege systems admit feasible interpolation, all bits of the secret key exchanged by the Diffie-Hellman protocol can be broken using a polynomial-size circuit.

The listed above results are conditional, and, therefore, it is still an open problem whether feasible interpolation can be applied to prove lower bounds for strong proof systems.

Extended resolution is a very powerful proof system that can simulate most of the standard proof systems including Frege and extended Frege systems. It has also exponential speed-up in comparison with weaker systems [3,4,5]. Thus, a lower bound on resolution refutations of pigeonhole principle is of exponential size, while there is a refutation of these class of formulas with extended resolution of polynomial size. Extended resolution involves a simple addition to the resolution system: It allows to add arbitrary lemmas to the formula being considered. At present no family of instances has been shown to be hard for extended resolution.

This paper demonstrates that an extended resolution refutation of CNFs representing the clique-graph colouring principle admits feasible interpolation. Since currently there are no results on the non-monotone circuit complexity, the main result of the paper is conditional : If there is a superpolynomial lower bound on the size of circuits interpolating the clique-graph colouring CNFs, then there is also a superpolynomial lower bound on extended resolution refutation for this class of formulas.

2 Background on Propositional Proof Systems

2.1 Preliminaries

Formulas in Conjunctive Normal Form (CNFs) are built from propositional *variables* from a set Var. A *literal* l is either a variable x or its negation $\neg x$, with $\mathsf{Var}(l) = x$ being the variable of l. A *clause* is a disjunction of literals, and a CNF is a conjunction of clauses.

We denote variables by x, y, z, literals by l, clauses by C, D and CNFs by φ, ψ. We use $\bot$ to refer to the empty clause and $\top$ to refer to the empty CNF. We denote a set of CNFs by CNF, a set of clauses by Cls, and a set of literals by Lit. We define $\mathsf{Cls}(\varphi)$ to be the set of clauses, $\mathsf{Lit}(\varphi)$ the set of literals, and $\mathsf{Var}(\varphi)$ the set of variables contained in φ.

An *assignment* is a function $\alpha : \mathsf{Var} \to \{1, 0\}$. We write $\alpha \models \varphi$ if φ evaluates to 1 for the assignment α, and we call α an *assignment satisfying* φ. We write $\alpha \not\models \varphi$ if φ evaluates to 0 for the assignment α, and we call α an *assignment falsifying* φ. A formula φ is *satisfiable* if there is an assignment satisfying it, and *unsatisfiable* otherwise. Given two formulas φ and ψ, we use $\varphi \sim \psi$ to denote that for each assignment α, $\alpha \models \varphi$ if and only if $\alpha \models \psi$.

2.2 Resolution and Extended Resolution

The *resolution* proof system, due to Robinson [12], consists of a single *resolution rule* that derives from two clauses containing a complementary literal a new clause as follows. Suppose we have $C \vee p$ and $D \vee \neg p$, where C and D are clauses and p is a propositional variable. The resolution rule allows us to deduce a new clause $C \vee D$.

Resolution is complete for propositional logic. A refutation of an unsatisfiable CNF φ starts with the clauses of φ and derives new clauses until a contradiction, represented by the empty clause, is obtained.

Definition 1 (Resolution refutation). *A resolution refutation of an unsatisfiable CNF $\varphi = C_1 \wedge \cdots \wedge C_n$ is a sequence of clauses $D_1, \ldots, D_m$ with the following properties.*

- *D_i a resolvent of two clauses from $\mathsf{Cls}(\varphi \wedge \bigwedge_{j \leq i-1} D_j)$.*
- *$D_m = \bot$ and $D_i \neq \bot$ for $i = 0, \ldots, m-1$.*

We say that m is the size of the resolution refutation.

Tseitin introduced the extension rule for the resolution calculus [13] as follows: If x, y and z are variables and x and $\neg x$ do not occur in any disjunction of a formula φ then φ can be extended with the clauses $x \vee y$, $x \vee z$ and $\neg x \vee \neg y \vee \neg z$.

In general, a new propositional variable y such that $y \notin \mathsf{Var}(\varphi)$, can be introduced by the extension rule as being equivalent to an arbitrary formula ψ such that $\mathsf{Var}(\psi) \subseteq \mathsf{Var}(\varphi)$.

Now $(y \leftrightarrow \psi)$ can be converted to a CNF φ^{ext} by De Morgan's laws (or by the Tseitin transformation), where $\varphi^{\mathsf{ext}} \sim (y \leftrightarrow \psi)$. Subsequently φ is replaced with a new formula $\varphi \wedge \varphi^{\mathsf{ext}}$. The rule can be applied consecutively an arbitrary number of times.

Definition 2 (Extension rule). *Let $\varphi = \varphi^y(\boldsymbol{x}, \boldsymbol{y}) \wedge \varphi^z(\boldsymbol{x}, \boldsymbol{z})$ be a CNF. The extension rule replaces φ with $\varphi \wedge \varphi^{\mathsf{ext}}$, where φ^{ext} is a CNF such that $\varphi^{\mathsf{ext}} \sim \bigwedge_{i \leq n} (v_i \leftrightarrow \psi_i)$ for some natural n and the following holds.*

- *$v_i \in \mathsf{Var} \backslash \mathsf{Var}(\varphi \wedge \bigwedge_{j \leq i-1} (v_j \leftrightarrow \psi_j))$.*
- *ψ_i is an arbitrary formula such that $\mathsf{Var}(\psi_i) \subseteq \mathsf{Var}(\varphi \wedge \bigwedge_{j \leq i-1} (v_j \leftrightarrow \psi_j))$.*

We say that φ^{ext} extends φ. The set of variables $\{v_1, \ldots, v_n\}$ is denoted by $\mathsf{Var}^{\mathsf{ext}}(\varphi)$.

In general, this rule trivially preserves satisfiability, but it does not preserve the validity, i.e. φ is unsatisfiable if and only if $\varphi \wedge \varphi^{\mathsf{ext}}$ is unsatisfiable.

Definition 3 (Extended resolution refutation). *We assume an unsatisfiable CNF φ. We say that a sequence of clauses $D_1^{\mathsf{ext}}, \ldots, D_l^{\mathsf{ext}}, D_1^{\mathsf{res}}, \ldots, D_m^{\mathsf{res}}$ is an extended resolution refutation of φ if the following holds.*

- *$\bigwedge_{i \leq l} D_i^{\mathsf{ext}}$ extends φ.*
- *$D_1^{\mathsf{res}}, \ldots, D_m^{\mathsf{res}}$ is a resolution refutation of $\varphi \wedge \bigwedge_{i \leq l} D_i^{\mathsf{ext}}$.*

We say that $l + m$ is the size of the refutation.

3 Feasible Interpolation

Feasible interpolation has been one of the most promising approaches for proving lower bounds for propositional proof systems and for bounded arithmetic using circuit lower bounds [9].

Craig's interpolation theorem for propositional logic states that if $\varphi_1 \to \varphi_2$ is valid, where φ_1 and φ_2 are propositional formulas then there is a formula ψ, an interpolant, such that both $\varphi_1 \to \psi$ and $\psi \to \varphi_2$ are valid [6,7].

Mundici [10] considered the question whether the interpolant of two propositional formulas of the form $\varphi_1 \to \varphi_2$ can always have a short circuit description, and showed that if this is the case then every problem in $\mathsf{NP} \cap \mathsf{co\text{-}NP}$ would have polynomial size circuits.

Let $\varphi^y(\boldsymbol{x}, \boldsymbol{y})$ and $\varphi^z(\boldsymbol{x}, \boldsymbol{z})$ be two propositional formulas having the occurrences of variables from the tuples $\boldsymbol{x} = (x_1, \ldots, x_n)$, $\boldsymbol{y} = (y_1, \ldots, y_s)$ and $\boldsymbol{z} = (z_1, \ldots, z_t)$. We assume that $\varphi^y(\boldsymbol{x}, \boldsymbol{y}) \wedge \varphi^z(\boldsymbol{x}, \boldsymbol{z})$ is an unsatisfiable formula. It implies that $\varphi^y(\boldsymbol{x}, \boldsymbol{y}) \to \neg\varphi^z(\boldsymbol{x}, \boldsymbol{z})$ is valid, and an interpolation function can be defined as follows.

Definition 4 (Interpolation function). *An interpolation function associated with a propositional formula $\varphi(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) = \varphi^y(\boldsymbol{x}, \boldsymbol{y}) \wedge \varphi^z(\boldsymbol{x}, \boldsymbol{z})$ is a Boolean function that outputs 0 only if $\varphi^y(\boldsymbol{\alpha}, \boldsymbol{y})$ is unsatisfiable, and outputs 1 only if $\varphi^z(\boldsymbol{\alpha}, \boldsymbol{z})$ is unsatisfiable for an assignment $\boldsymbol{\alpha}$ of the variables $\boldsymbol{x}$.*

As it is mentioned above, interpolation functions are not always computable in polynomial time unless $\mathsf{P} = \mathsf{NP} \cap \mathsf{co\text{-}NP}$ [10]. Now we define a feasible interpolation property.

Definition 5 (Feasible interpolation property). *A proof system $\mathcal{P}$ admits feasible interpolation if whenever $\mathcal{P}$ has a polynomial-size refutation of a formula φ, there is an interpolation function associated with φ such that it has a polynomial-size circuit.*

The idea of feasible interpolation works as follows [9]. We establish for a given proof an upper bound on the computational complexity of an interpolant of $\varphi^y(\boldsymbol{x}, \boldsymbol{y})$ and $\varphi^z(\boldsymbol{x}, \boldsymbol{z})$ in terms of the size of a proof of the unsatisfiability of $\varphi^y(\boldsymbol{x}, \boldsymbol{y}) \wedge \varphi^z(\boldsymbol{x}, \boldsymbol{z})$. Then any pair $\varphi^y(\boldsymbol{x}, \boldsymbol{y})$ and $\varphi^z(\boldsymbol{x}, \boldsymbol{z})$ which is hard to interpolate yields a formula that must have large proofs of the unsatisfiability in the proof system P. Since lower bounds on a circuit size are known only for the monotone case, we obtain unconditional lower bounds by considering a monotone version of the above idea.

4 The Karchmer-Wigderson Game

In the following we use a notion of communication complexity introduced by Yao [14]. We consider two disjoint sets $\mathcal{U}_n$ and $\mathcal{V}_n$ of n-bit strings and two players. The U-player receives an n-bit string $\boldsymbol{u} \in \mathcal{U}_n$ and the V-player another n-bit string $\boldsymbol{v} \in \mathcal{V}_n$. Then the Karchmer-Wigderson game corresponding to a nonconstant Boolean function $f : \{0,1\}^n \to \{0,1\}$ is as follows.

- U-player gets an input $\boldsymbol{u} \in \{0,1\}^n$ such that $f(\boldsymbol{u}) = 1$.
- V-player gets an input $\boldsymbol{v} \in \{0,1\}^n$ such that $f(\boldsymbol{v}) = 0$.
- The goal of the game is for the U-player and the V-player to agree on an index $i \in [n]$ such that $u_i \neq v_i$.

Before the game starts, the players agree on a protocol for exchanging messages. They have to minimize over all protocols the number of bits they have to communicate in the worst case. This minimum is called the communication complexity of the game.

We define protocol for the Karchmer-Wigderson game on $\mathcal{U}_n$ and $\mathcal{V}_n$ formally as in [9]. It consists of the following components.

1. A directed acyclic graph $\mathsf{G} = (\mathsf{Nodes}, \mathsf{Edges})$, where Nodes is a set of nodes and Edges is a set of edges.
2. A strategy function, Str, which takes as an input a triple $(p, \boldsymbol{u}, \boldsymbol{v})$, where $p \in \mathsf{Nodes}$, $\boldsymbol{u} \in U$ and $\boldsymbol{v} \in V$ and outputs a node $q \in \mathsf{Nodes}$ such that $(p, q) \in \mathsf{Edges}$.
3. A subset of Nodes called the consistency condition for $(\boldsymbol{u}, \boldsymbol{v})$ and denoted by $\mathsf{Cons}(\boldsymbol{u}, \boldsymbol{v})$.

These components satisfy the following conditions.

1. G has one node with in-degree 0, called the source and labeled by $\emptyset$. There are two kinds of nodes. The nodes with the out-degree 0 are leaves and other nodes are inner nodes.
2. Each leaf is labeled either by $u_i = 1 \wedge v_i = 0$ or by $u_i = 0 \wedge v_i = 1$ for $1 \leq i \leq n$.
3. For every pair $\boldsymbol{u} \in \mathcal{U}_n$, $\boldsymbol{v} \in \mathcal{V}_n$, the consistency condition $\mathsf{Cons}(\boldsymbol{u}, \boldsymbol{v})$ has the properties that it contains the source node and the labels of the leaves in $\mathsf{Cons}(\boldsymbol{u}, \boldsymbol{v})$ are valid for (u, v).
4. For each $p \in \mathsf{Cons}(\boldsymbol{u}, \boldsymbol{v})$, the path $\Pi^p_{\boldsymbol{u},\boldsymbol{v}}$ in G, which starts at p and is determined by the strategy function Str with input $(\boldsymbol{u}, \boldsymbol{v})$, is contained in $\mathsf{Cons}(\boldsymbol{u}, \boldsymbol{v})$.

A protocol is called *monotone* if every leaf in it is labeled by one of the formulas $u_i = 1 \wedge v_i = 0$ for $1 \leq i \leq n$.

The *communication complexity* of $\mathsf{G} = (\mathsf{Nodes}, \mathsf{Edges})$ is the minimal number θ such that for every $p \in \mathsf{Nodes}$ the players decide whether $p \in \mathsf{Cons}(\boldsymbol{u}, \boldsymbol{v})$ and compute $\mathsf{Str}(p, \boldsymbol{u}, \boldsymbol{v})$ with at most θ bits exchanged in the worst case.

The following theorem can be used to determine an upper bound on the size of a circuit-interpolant.

Theorem 6 (Razborov [11], Krajíček [9]). *Let $U, V \subseteq \{0,1\}^n$ be two disjoint sets. Let G be a protocol for the Karchmer-Wigderson game on U and V which has k nodes and the communication complexity θ. Then there is a circuit $\mathbb{C}$ of size $k2^{O(\theta)}$ separating U from V. Moreover, if G is monotone so is $\mathbb{C}$.*

5 Clique-Colouring Principle

In order to prove a superpolynomial lower bound one needs two CNFs that are hard to interpolate. We use unsatisfiable CNFs based on the clique-colouring principle. This principle expresses the fact that if there is a graph with a clique of size k, then any colouring of it requires at least k colours. The clique-colouring principle can be seen as a form of the pigeonhole principle since it represents that the vertices in a clique must all have distinct colours.

Definition 7 (Clique-colouring principle). *Let n, s, t be natural numbers and let $\binom{n}{2}$ to denote the set of two-element subsets of $\{1, 2, \ldots, n\}$.*

1. The set $\mathsf{Clique}_{n,s}(\boldsymbol{x}, \boldsymbol{y})$ is a set of the formulas in the variables x_{ij}, $\{i, j\} \in \binom{n}{2}$, and y_{kl} for $1 \le k \le s$, $1 \le l \le n$.

(1a) $\bigvee_{l \le n} y_{kl}$ for all $k \le s$.

(1b) $\neg y_{kl} \vee \neg y_{k'l}$ for $k < k' \le s$ and $1 \le l \le n$.

(1c) $\neg y_{ki} \vee \neg y_{k'j} \vee x_{ij}$ for $k < k' \le s$ and $\{i, j\} \in \binom{n}{2}$.

2. The set $\mathsf{Colour}_{n,t}(\boldsymbol{x}, \boldsymbol{z})$ is a set of the formulas in the variables x_{ij}, $\{i, j\} \in \binom{n}{2}$, and z_{il} for $1 \le i \le n$ and $1 \le l \le t$.

(2a) $\bigvee_{l \le t} z_{il}$ for all $i \le n$.

(2b) $\neg z_{il} \vee \neg z_{il'}$ for $l < l' \le t$ and $1 \le i \le n$.

(2c) $\neg z_{il} \vee \neg z_{jl} \vee \neg x_{ij}$ for all $l \le t$ and $\{i, j\} \in \binom{n}{2}$.

The formula $\mathsf{Clique}_{n,s}(\boldsymbol{x}, \boldsymbol{y})$ consists of all formulas in $1a - 1c$, and the formula $\mathsf{Colour}_{n,t}(\boldsymbol{x}, \boldsymbol{z})$ consists of all formulas in $2a - 2c$. In the following we will use $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ as a shortcut for $\mathsf{Clique}_{n,s}(\boldsymbol{x}, \boldsymbol{y}) \wedge \mathsf{Colour}_{n,t}(\boldsymbol{x}, \boldsymbol{z})$. The formula $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ is trivially unsatisfiable for $s > t$.

Given $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$, we denote by X the set of x variables, and by $\tilde{n}$, $\tilde{s}$ and $\tilde{t}$ the number of x, y and z variables correspondingly.

Let $\overline{\mathsf{Cls}} = \{C_1, \ldots, C_{\tilde{n}}\}$ be a subset of the set of clauses of $\mathsf{Clique}_{n,s}(\boldsymbol{x}, \boldsymbol{y})$ and it is defined as follows.

1. For any $C \in \overline{\mathsf{Cls}}$ there is $x \in \mathsf{X}$ such that $x \in \mathsf{Lit}(C)$.
2. $\mathsf{Var}(C_i) \cap \mathsf{Var}(C_j) \cap \mathsf{X} = \emptyset$ for distinct $C_i, C_j \in \overline{\mathsf{Cls}}$.

Note that $\overline{\mathsf{Cls}}$ is not unique in general. In the following we assume that the players agree on some unique choice of $\overline{\mathsf{Cls}}$ before they start constructing a protocol for the Karchmer-Wigderson game.

6 Interpolation Theorem

Now we show that an extended resolution refutation of $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ admits a form of feasible interpolation. The proof of the theorem is based on constructing a protocol for the Karchmer-Wigderson game. The main difference with the standard approach that we construct the game for a restricted set of pairs $(\boldsymbol{u}, \boldsymbol{v})$, where $(\boldsymbol{u}, \boldsymbol{q})$ is an assignment satisfying $\mathsf{Clique}_{n,s}(\boldsymbol{x}, \boldsymbol{y})$ for some $\boldsymbol{q}$, and $(\boldsymbol{v}, \boldsymbol{r})$ is an assignment satisfying $\mathsf{Colour}_{n,t}(\boldsymbol{x}, \boldsymbol{z})$ for some $\boldsymbol{r}$. Later we demonstrate that this restriction is eligible.

We consider the following sets of assignments of variables contained in CNFs $\mathsf{Clique}_{n,s}(\boldsymbol{x}, \boldsymbol{y})$ and $\mathsf{Colour}_{n,t}(\boldsymbol{x}, \boldsymbol{z})$.

$$\mathcal{U}_n = \{\boldsymbol{u} \in \{0,1\}^{\tilde{n}} \mid \exists \boldsymbol{q} \in \{0,1\}^{\tilde{s}} : (\boldsymbol{u}, \boldsymbol{q}) \models \mathsf{Clique}_{n,s}(\boldsymbol{x}, \boldsymbol{y})\} \text{ and}$$

$$\mathcal{V}_n = \{\boldsymbol{v} \in \{0,1\}^{\tilde{n}} \mid \exists \boldsymbol{r} \in \{0,1\}^{\tilde{t}} : (\boldsymbol{v}, \boldsymbol{r}) \models \mathsf{Colour}_{n,t}(\boldsymbol{x}, \boldsymbol{z})\} \ .$$

Now we define $\overline{\mathcal{U}}_n, \overline{\overline{\mathcal{U}}}_n \subseteq \mathcal{U}_n$, and $\overline{\mathcal{V}}_n, \overline{\overline{\mathcal{V}}}_n \subseteq \mathcal{V}_n$ as follows.

1. $\overline{\mathcal{U}}_n \cup \overline{\overline{\mathcal{U}}}_n = \mathcal{U}_n$ and $\overline{\mathcal{V}}_n \cup \overline{\overline{\mathcal{V}}}_n = \mathcal{V}_n$.
2. $\sum_{x \in \mathsf{X}} \alpha(x) = 2k + 1$, $k \in \mathbb{N}$, for each $\alpha \in \overline{\mathcal{U}}_n$.
3. $\sum_{x \in \mathsf{X}} \alpha(x) = 2k$, $k \in \mathbb{N} \cup \{0\}$, for each $\alpha \in \overline{\overline{\mathcal{U}}}_n$.
4. $\sum_{x \in \mathsf{X}} \alpha(x) = 2k + 1$, $k \in \mathbb{N}$, for each $\alpha \in \overline{\mathcal{V}}_n$.
5. $\sum_{x \in \mathsf{X}} \alpha(x) = 2k$, $k \in \mathbb{N} \cup \{0\}$, for each $\alpha \in \overline{\overline{\mathcal{V}}}_n$.

We continue with a combinatorial lemma we need to prove Theorem 9.

Lemma 8. *Let $S = \{s_1, \ldots, s_k\}$ be a set and $P : S \to \{1, 0\}$ be a predicate. Suppose $S^t = \{s \in S \mid P(s) = 1\}$ and $S^f = \{s \in S \mid P(s) = 0\}$. If $|S^t| > |S^f|$ then for arbitrary sets S_1 and S_2 such that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$ at least one of the following holds.*

1. $|S_1^t| > |S_1^f|$.
2. $|S_2^t| > |S_2^f|$.

Proof. By contradiction. Assume that $|S_1^t| \leq |S_1^f|$ and $|S_2^t| \leq |S_2^f|$. Then

$$|S^t| = S_1^t| + S_2^t| \leq S_1^f| + S_2^f| = |S^f| \ ,$$

and we obtain a contradiction. □

In general a choice of S_1 and S_2 in Lemma 8 is not unique. When we need to apply the lemma, we assume that an order on clauses is defined and based on this order a unique choice of S_1 and S_2 can be made.

Now we prove a theorem showing a kind of feasible interpolation for each extended resolution refutation of $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$. Theorem 9 in combination with Lemma 11 in Section 7 implies that feasible interpolation holds for these derivations.

Theorem 9. *Suppose $\mathcal{D} = D_1^{\mathsf{ext}}, \ldots, D_L^{\mathsf{ext}}, D_1^{\mathsf{res}}, \ldots, D_K^{\mathsf{res}}$ is an extended resolution refutation of $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ for $n \in \mathbb{N}$ and $t < s$. Then there is a circuit separating the sets $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$ (resp. the sets $\overline{\overline{\mathcal{U}}}_n$ and $\overline{\mathcal{V}}_n$) of size at most $Kn^{O(1)}$.*

Proof. Let the U-player know $\boldsymbol{u} = (u_1, \ldots, u_n) \in \overline{\mathcal{U}}_n$ and the V-player know $\boldsymbol{v} = (v_1, \ldots, v_n) \in \overline{\overline{\mathcal{V}}}_n$. The players agree on a unique choice of $\overline{\mathsf{Cls}}$ before they start a game. They compute a label $\mathsf{L}(C)$ for each $C \in \mathcal{D}$ recursively and independently according the protocol defined before the game. The labels are subsets of clauses from the set $\overline{\mathsf{Cls}}$ satisfying the following.

1. $\mathsf{L}(D_K^{\mathsf{res}}) = \overline{\mathsf{Cls}}$.
2. If C is derived from C_1 and C_2 and $\mathsf{L}(C) = G$ then $\mathsf{L}(C_1), \mathsf{L}(C_2) \subseteq G$, $\mathsf{L}(C_1) \cup \mathsf{L}(C_2) = G$ and $\mathsf{L}(C_1) \cap \mathsf{L}(C_2) = \emptyset$.

It is trivially possible to define a protocol for choosing unique labels if to introduce an order on clauses. The players can compute the labels independently, i.e. without sending bits of information each other.

Predicates $\mathsf{P}^u, \mathsf{P}^v : \mathcal{D} \to \{1, 0\}$ are defined as follows. Let $\mathsf{L}(C) = G$. Then $\mathsf{P}^u(C) = 1$ (resp. $\mathsf{P}^v(C) = 1$) if for $W = \{i \in \mathbb{N} \mid \exists D \in G : x_i \in \mathsf{Var}(D)\}$

$$\sum_{i \in W} u_i > \sum_{i \in W} v_i$$

(resp. $\sum_{i \in W} u_i < \sum_{i \in W} v_i$). Otherwise $\mathsf{P}^u(C) = 0$ (resp. $\mathsf{P}^v(C) = 0$).

By definition of $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$, $\sum_{1 \le i \le n} u_i \neq \sum_{1 \le i \le n} v_i$ for $\boldsymbol{u} = (u_1, \ldots, u_n) \in \overline{\mathcal{U}}_n$ and $\boldsymbol{v} = (v_1, \ldots, v_n) \in \overline{\overline{\mathcal{V}}}_n$. Hence either $\mathsf{P}^u(D_K^{\mathsf{res}}) = 1$ or $\mathsf{P}^v(D_K^{\mathsf{res}}) = 1$. By Lemma 8, if C is derived from C_1 and C_2 and $\mathsf{P}^u(\mathsf{L}(C)) = 1$ then $\mathsf{P}^u(\mathsf{L}(C_1)) \vee \mathsf{P}^u(\mathsf{Label}(C_2)) = 1$. The same property holds for P^v.

Now we estimate the computational complexity of computing $\mathsf{P}^u(C)$ and $\mathsf{P}^v(C)$ for each C. We observe that $\tilde{n} = O(n^2)$ and hence $|\overline{\mathsf{Cls}}| = O(n^2)$ and $\sum_{i \in W} u_i \le O(n^2)$, $\sum_{i \in W} v_i \le O(n^2)$. Thus it is sufficient for the players to send each other $O(\log n^2)$ bits.

The players construct a path $\pi = \pi_0, \ldots, \pi_l$ through the given refutation such that the following holds.

1. $\pi_0 = D_K^{\mathsf{res}}$.
2. π_{a+1} is one of two clauses which are the hypotheses of the inference yielding π_a for $1 \le a \le K$.
3. $\pi_a \in \{D_1^{\mathsf{res}}, \ldots, D_K^{\mathsf{res}}\}$ for $1 \le a < l$.
4. $l \le K + 1$.

By the definition of sets $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$ either $\mathsf{P}^u(D_K^{\mathsf{res}}) = 1$ or $\mathsf{P}^v(D_K^{\mathsf{res}}) = 1$. Let $\pi_a = C$ was inferred from C_1 and C_2. Then the following outcomes are possible.

1. $\mathsf{P}^u(C) \wedge \mathsf{P}^u(C_1) = 1$ or $\mathsf{P}^v(C) \wedge \mathsf{P}^v(C_1) = 1$.
2. $\mathsf{P}^u(C) \wedge \mathsf{P}^u(C_2) = 1$ or $\mathsf{P}^v(C) \wedge \mathsf{P}^v(C_2) = 1$.

In the first case the players choose $\pi_{a+1} = C_1$, otherwise they choose $\pi_{a+1} = C_2$. Note that we have chosen $\overline{\mathsf{Cls}}$ as a subset of clauses in $\mathsf{Clique}_{n,s}(\boldsymbol{x}, \boldsymbol{y})$. If $\pi_a \in \overline{\mathsf{Cls}}$ the players stop to construct the path and find $i \le n$ such that $u_i \neq v_i$. By construction either $\mathsf{P}^u(\pi_a) = 1$ or $\mathsf{P}^v(\pi_a) = 1$. It means that is an i such that $x_i \in \mathsf{Var}(C)$ and $u_i \neq v_i$.

Now we define the protocol formally as follows.

1. G has $K + 2\tilde{n}$ nodes, where K is the number of clauses in the refutation and $2\tilde{n}$ nodes labeled with $u_i = 1 \wedge v_i = 0$ and $u_i = 0 \wedge v_i = 1$ for $1 \leq i \leq \tilde{n}$.
2. The consistency condition $\mathsf{Cons}(\boldsymbol{u}, \boldsymbol{v})$ contains a clause $D \in \{D_1^{\mathsf{res}}, \dots, D_K^{\mathsf{res}}\}$ if $\mathsf{P}^u(D) = 1$ or $\mathsf{P}^v(D) = 1$.
3. The strategy function Str is defined as follows.
   (a) If $\mathsf{P}^u(D) = 1$ and D is inferred from C_1 and C_2 then
   $$\mathsf{Str}(\boldsymbol{u}, \boldsymbol{v}, D) = \begin{cases} C_1, \text{ if } \mathsf{P}^u(C_1) = 1 \\ C_2, \text{ otherwise} \end{cases}$$
   (b) If $\mathsf{P}^v(D) = 1$ and D is inferred from C_1 and C_2 then
   $$\mathsf{Str}(\boldsymbol{u}, \boldsymbol{v}, D) = \begin{cases} C_1, \text{ if } \mathsf{P}^v(C_1) = 1 \\ C_2, \text{ otherwise} \end{cases}$$
   (c) If $\pi_a \in \overline{\mathsf{Cls}}$ the players find $i \leq \tilde{n}$ such that $u_i \neq v_i$. In this case $\mathsf{Str}(\boldsymbol{u}, \boldsymbol{v}, D_j)$ is one of the nodes labeled with $u_i = 1 \wedge v_i = 0$ and $u_i = 0 \wedge v_i = 1$.

As the protocol has $K + 2\tilde{n}$ nodes and for each node the players exchange at most $\log(n^2)$ bits, Theorem 6 yields that circuits separating $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$ have size at most $Kn^{O(1)}$. Analogously we obtain the same upper bound on the size of circuits separating the sets $\overline{\overline{\mathcal{U}}}_n$ and $\overline{\mathcal{V}}_n$. □

7 Main Result

The main result is that the existence of a superpolynomial lower bound on the size of a non-monotone circuit for clique functions implies a superpolynomial lower bound on the size of extended resolution refutations of CNFs representing the clique-colouring principle.

The clique function is NP-complete. Hence we suppose that circuits for computing it cannot have polynomial size. For monotone circuits we can prove an exponential lower bound. Thus Alon and Boppana [1] have proved exponential lower bounds on the size of monotone circuits which separate the clique-colouring pair.

Theorem 10 (Alon-Boppana [1]). *Assume that $t = \lceil \sqrt{n} \rceil$ and $s = t + 1$. Then $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ has a monotone interpolation circuit of size at least $2^{\Omega(n^{1/4})}$ for sufficiently large n.*

At present there are no results on superpolynomial lower bounds for non-monotone circuits.

Now we define the relation between the complexity of circuits interpolating $\mathcal{U}_n$ and $\mathcal{V}_n$ and the complexity of circuits interpolating $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$.

Lemma 11. *Let* $t = \lceil \sqrt{n} \rceil$ *and* $s = t+1$. *Assume that* $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ *has no interpolation circuit smaller than* $\Omega(f(n))$ *for a superpolynomial function* $f : \mathbb{N} \to \mathbb{N}$. *Then there is no interpolation circuit for* $\overline{\mathcal{U}}_n$ *and* $\overline{\overline{\mathcal{V}}}_n$ *smaller than* $f(n)^{\Omega(1)}$.

Proof. The size of the smallest interpolation circuit for $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ is polynomial in the size of the smallest interpolation circuit for $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$. This implies that if there is an interpolation circuit for $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$ smaller than $f(n)^{\Omega(1)}$ then there is an interpolation circuit for $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ smaller than $\Omega(f(n))$. This is a contradiction, and there is no interpolation circuit for $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$ smaller than $f(n)^{\Omega(1)}$.

□

Now we can derive a conditional lower bound on extended resolution refutation for a clique-colouring principle.

Theorem 12. *Let* $t = \lceil \sqrt{n} \rceil$ *and* $s = t+1$. *Suppose* $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ *has no interpolation circuit smaller than* $\Omega(f(n))$ *for a superpolynomial function* $f(n) : \mathbb{N} \to \mathbb{N}$. *Then there is a superpolynomial function* $g : \mathbb{N} \to \mathbb{N}$ *such that* $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ *has no extended resolution refutation smaller than* $\Omega(g(n))$.

Proof. Suppose there is an extended resolution refutation of the clique-colouring principle $\mathsf{CliqueColour}_{n,s,t}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ of size K. By Theorem 9 there is a circuit interpolating the sets $\overline{\mathcal{U}}_n$ and $\overline{\overline{\mathcal{V}}}_n$ of size at most $Kn^{O(1)}$. By Lemma 11 and the theorem assumptions we have $Kn^{O(1)} \geq f(n)^{\Omega(1)})$ and therefore there is a superpolynomial function $g : \mathbb{N} \to \mathbb{N}$ such that $K \geq \Omega(g(n))$.

□

8 Conclusions

Extended resolution is a powerful proof system which is polynomially equivalent to Frege systems and extended Frege systems. It is still unknown whether the most powerful proof systems have superpolynomial lower bounds. Another major open problem is to understand which methods can be used to prove lower bounds for these systems. One of the prominent approaches for proving lower bounds is based on feasible interpolation. There are some attempts to clarify its suitability for strong proof systems but only conditional results are known so far. However, the non-applicability of feasible interpolation for proving lower bounds for strong proof systems does not follow immediately from disproving this property in general. What we need, in fact, is just a class of formulas hard to interpolate and for which extended resolution refutations admit feasible interpolation. This paper investigates the applicability of feasible interpolation for proving lower bounds and it demonstrates the following.

1. An extended resolution refutation of CNFs representing the clique-colouring principle admits the feasible interpolation property.

2. The existence of a superpolynomial lower bound on the size of an interpolation circuit for the clique-colouring pair implies a superpolynomial lower bound on the size of an extended resolution refutation for this class of formulas.

Since the clique function is NP-complete, it is expected that circuits for computing it cannot have polynomial size, but currently there are no results on superpolynomial lower bounds for non-monotone circuits.

It remains an open problem whether an extended resolution refutation of the clique-colouring principle admits the monotone version of feasible interpolation.

References

1. Alon, N., Boppana, R.: The monotone circuit complexity of boolean functions. Combinatorica 7(1), 1–22 (1987)
2. Bonet, M.L., Pitassi, T., Raz, R.: No feasible interpolation for TC^0-Frege proofs. In: FOCS, pp. 254–263 (1997)
3. Cook, S.: Feasibly constructive proofs and the propositional calculus (preliminary version). In: Proceedings of the 7th Annual ACM Symposium on Theory of Computing (STOC), pp. 83–97 (1975)
4. Cook, S.: A short proof of the pigeon hole principle using extended resolution. ACM SIGACT News 8(4), 28–32 (1976)
5. Cook, S., Reckhow, R.: On the lengths of proofs in the propositional calculus (preliminary version). In: STOC, pp. 135–148 (1974)
6. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. Journal of Symbolic Logic 22(3), 250–268 (1957)
7. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. Journal of Symbolic Logic 22(3), 269–285 (1957)
8. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
9. Krajícek, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. The Journal of Symbolic Logic 62(2), 457–486 (1997)
10. Mundici, D.: A lower bound for the complexity of Craig's interpolants in sentential logic. Archiv. Math. Logik 23, 27–36 (1983)
11. Razborov, A.A.: Unprovability of lower bounds on circuit size in certain fragments of bounded arithmetic. Izv. RAN. Ser. Mat. 59, 201–224 (1995)
12. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM (JACM) 12(1), 23–41 (1965)
13. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Leningrad Seminar on Mathematical Logic (1970)
14. Yao, A.C.: Some complexity questions related to distributive computing (preliminary report). In: STOC, pp. 209–213 (1979)

A Turing Machine Distance Hierarchy

Stanislav Žák[*] and Jiří Šíma[**]

Institute of Computer Science, Academy of Sciences of the Czech Republic,
P. O. Box 5, 18207 Prague 8, Czech Republic
{stan,sima}@cs.cas.cz

Abstract. We introduce a new so-called distance complexity measure for Turing machine computations which is sensitive to long-distance transfers of information on the worktape. An important special case of this measure can be interpreted as a kind of buffering complexity which counts the number of necessary block uploads into a virtual buffer on top of the worktape. Thus, the distance measure can be used for investigating the buffering aspects of Turing computations. In this paper, we start this study by proving a tight separation and hierarchy result. In particular, we show that a very small increase in the distance complexity bound (roughly from $c(n)$ to $c(n + 1) + \mathit{constant}$) brings provably more computational power to both deterministic and nondeterministic Turing machines. For this purpose, we formulate a very general diagonalization method for Blum-like complexity measures. We also obtain a hierarchy of the distance complexity classes.

Keywords: Turing machine, hierarchy, distance complexity, diagonalization

1 Introduction

The theory of computational complexity is one of the major attempts to understand the phenomenon of computation. One of the key tasks of the theory is to find out how an increase or decrease of limits set on the computational resources can influence the computational power of different types of computational devices. In history, the efforts to answer questions of this type led to a long sequence of various separation and hierarchy results for particular computational devices and complexity measures, e.g. chronologically [3,6,7,8,9,1,4,5].

The present paper follows this direction of research. A new nontraditional complexity measure is introduced for both deterministic and nondeterministic Turing machines (TM) with one worktape (Section 2). This so-called *distance complexity* (Section 6) is sensitive to long transfers of information on the worktape of a Turing machine while the short transfers are not counted.

[*] S.Ž.'s research was partially supported by the projects GA ČR P202/10/1333 and RVO: 67985807.

[**] J.Š.'s research was partially supported by the projects GA ČR P202/12/G061 and RVO: 67985807.

In particular, the distance complexity of a given computation, which can be understood as a sequence of TM configurations, is the length of its certain subsequence defined as follows. The subsequence starts with the initial configuration c_0. The next element c_{i+1} is defined to be the first configuration c after c_i such that the distance (measured in the number of tape cells) between the worktape head position of c and the worktape head position of some configuration that precedes c and succeeds or is equal to c_i, reaches a given bound $d(n)$ where $d(n)$ is a parameter of the distance measure depending on the input length n. In other words, c_{i+1} is the first configuration along the computation such that its worktape head position is exactly at distance $d(n)$ from that of some preceding configuration taken from the segment of computation starting with c_i.

A special case of this distance complexity in which the distance is measured from the head position of previous-element configuration c_i (i.e. the worktape head position of c_{i+1} is at distance $d(n)$ from that of c_i), can be interpreted as a kind of *buffering complexity* of Turing computations. In particular, the worktape is divided into blocks of the same number $d(n)$ of cells and the control unit has a virtual buffer memory whose capacity is two blocks, the concatenated so-called left-hand and right-hand side buffers, respectively. The worktape head position has to be within this buffer and the buffering complexity measures the number of necessary block uploads into the buffer. Namely, if the worktape head finds itself at the right end of the right-hand side buffer containing the $(k+1)$st block of the worktape (while the left-hand side buffer stores the kth worktape block), and has to move further to the right worktape cell $(k+1)d(n)+1$ outside the buffer, then the content of the right-hand side buffer is copied to the left-hand side one and the $(k+2)$nd block of the worktape is uploaded to the right-hand side buffer while the worktape head ends up at the left end of the right-hand side buffer. In other words, the virtual buffer moves to the right by $d(n)$ cells which is reminiscent of a super-head reading the whole block as a super-cell of length $d(n)$. Similarly, for the worktape head at the left end of the left-hand side buffer which moves to the left. In this way, the distance measure can be used for investigating the buffering aspects of Turing computations.

We start our study by separation (Section 6) and hierarchy (Section 7) results for the distance complexity which are surprisingly very tight. This indicates that the new complexity measure is an appropriate tool for classifying the computations. The tightness in the results requires that the worktape alphabet is fixed and the measure is not applied to TM computations directly but instead to their simulations on a fixed universal Turing machine. The results are of the form that a shift by one in the argument of the complexity bound (and of parameter d plus an additive constant) leads to a strictly greater computational power. In the case of a linear complexity bound, the increase in the bound by an additive constant is sufficient to gain more power. For the hierarchy of complete languages the increase in the bound is slightly larger (Section 7). The main tool of the proof is the general diagonalization method introduced in [9] (Section 3) which is applied (Section 5) to arbitrary Blum-like complexity measures (Section 4).

2 Technical Preliminaries

We denote by N the set of natural numbers (including 0). By a complexity bound we mean any mapping c, $c : N \to N$. In the notation of complexity classes, $c(n+1)$ stands for complexity bound c' such that, for each $n \in N$, $c'(n) =_{df} c(n+1)$. By a language we mean any L, $L \subseteq \{0,1\}^*$. Moreover, ϵ denotes the empty word.

By a *Turing machine* we mean any machine with two-way read-only input tape and with one semi-infinite worktape (infinite to the right) with worktape alphabet $0, 1, b$ and with endmarker # at the left end side (at the 0-th cell of the tape), allowing for both the deterministic or nondeterministic versions. Any computation of a Turing machine on an input word can be understood as a sequence of its configurations.

The programs are words from $\{0,1\}^+$ and we suppose that there is a machine which, having any word p on its worktape, is able to decide whether p is a program without any use of the cells outside of p. If p is a program, then M_p is the corresponding machine. For any machine M, by $L(M)$ we denote the language accepted by M, and by p_M we mean the program of M.

On any input u, the universal machine starts its computation with some program p at the left end side of the worktape and it simulates machine M_p on u. We implicitly assume that the universal machine shifts program p along its worktape in order to follow the shifts of the head of the simulated machine M_p.

Let S be a set of programs and let $C = \{L_p \,|\, p \in S\}$ be a class of languages. We say that C is uniformly recursive iff there is a machine M such that for each $p \in S$ and for each $u \in \{0,1\}^*$, computing on the input pu, M decides whether $u \in L_p$ or not.

3 The Diagonalization Theorem

In the sequel we use the diagonalization method which is based on the theorem from [9]. This theorem is formulated without any notion concerning computability nor complexity. It is formulated only in terms of languages, functions and relations. Due to this property the method is largely applicable towards the world of computational complexity.

We say that two languages L, L' are equivalent $L \sim L'$ iff they differ only on a finite number of words. For a class C of languages we define $\mathcal{E}(C) =_{df} \{L' \,|\, \exists L \in C\ (L \sim L')\}$. Then the fact $L \notin \mathcal{E}(C)$ implies that L differs from any language of $\mathcal{E}(C)$ on infinitely many words.

Now we are ready to introduce our diagonalization theorem.

Theorem 1. *Let L be a language and let C be a class of languages indexed by a set S, that is $C = \{L_p \,|\, p \in S\}$. Let R be a language and let F be a mapping, $F : R \to S$, such that $(\forall\ p \in S)(\exists^\infty\ r \in R)(F(r) = p)$. Let z be a mapping, $z : R \to N$, such that for each $r \in R$, $z(r)$ satisfies the following two conditions:*
a) $r1^{z(r)} \in L \ \leftrightarrow\ r \notin L_{F(r)}$,
b) $(\forall j,\, 0 \le j < z(r))(r1^j \in L \leftrightarrow\ r1^{j+1} \in L_{F(r)})$.
Then $L \notin \mathcal{E}(C)$.

The idea of the proof by contradiction is as follows. From the assumption $L \in \mathcal{E}(C)$ we derive an appropriate r such that $L = L_{F(r)}$. Then conditions (a), (b) produce a contradiction immediately.

The idea of the application to the complexity world is as follows. The decision whether $r \in L_{F(r)}$ or not is achieved during the computation on the word $r1^{z(r)}$ where $z(r)$ is a very large function. While from the point of length $|r|$ this decision requires very large amount of the source in question (e. g. space, time etc.), especially in the case of nondeterministic computations, from the point of length $|r1^{z(r)}|$ this amount is negligible. The main consumption of the sources is now concentrated in the simulation on the input of length $n + 1$. Even in the case of nondeterministic computations the respective increase of complexity is moderate.

For the sake of completeness we add the complete proof of the theorem [9].

Proof. By contradiction. Let $L \in \mathcal{E}(C)$. Hence, $L \sim L_p$ for some $p \in S$. Moreover, there is $r \in R$ such that $F(r) = p$ and the languages L and $L_{F(r)}(= L_p)$ differ only on words shorter than r. In particular, for each $j \in N$, $r1^j \in L_{F(r)}$ iff $r1^j \in L$. Hence by condition (b), $r \in L \leftrightarrow r1^{z(r)} \in L$, and further by condition (a), $r1^{z(r)} \in L \leftrightarrow r \notin L$, which is a contradiction. $\square$

4 Complexity Measures and Classes

Inspired by Blum axioms [2], by a complexity measure we mean any (partial) mapping $m : \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \to N$. Informally, the first argument is intended to be the input word, the second one corresponds to a program, and the third one represents the initial content of the worktape.

Let S, $S \subseteq \{0,1\}^+$, be the set of programs of the machines in question. For any $p \in S$ and for any complexity bound $c : N \to N$ we define $L_{m,p,c} =_{df} \{u \in \{0,1\}^* | m(u,p,\epsilon) \leq c(|u|)\}$ where ϵ is the empty word. We say that M_p accepts its language within m-complexity c iff $L(M_p) = L_{m,p,c}$.

Let m be a complexity measure, U be a universal machine, and p_U be the program of U. By the complexity measure m_U we mean the mapping $m_U : \{0,1\}^* \times \{0,1\}^* \to N$, $m_U(u,p) =_{df} m(u,p_U,p)$.

For any $p \in S$ and for any $c : N \to N$ we define language $L_{m_U,p,c} =_{df} \{u \in \{0,1\}^* \,|\, m_U(u,p) \leq c(|u|)\}$. We say that M_p accepts its language within m_U-complexity c iff $L(M_p) = L_{m_U,p,c}$. We also say that $L(M_p)$ is an (m_U, c)-complete language.

We define the complexity class $C_{m,c} =_{df} \{L_{m,p,c} \,|\, p \in S\}$. Similarly for $m = m_U$, $C_{m_U,c} =_{df} \{L_{m_U,p,c} \,|\, p \in S\}$. We say that M_p accepts its language within m-complexity c iff $L(M_p) = L_{m,p,c}$.

Let $C_{comp,m_U,c} =_{df} \{L \,|\, (\exists p \in S)\ L = L(M_p) = L_{m_U,p,c}\}$ be a class composed of all (m_U, c)-complete languages which we call an (m_U, c)-complete (or shortly complete) class.

5 The Diagonalization Result

The following definition forms the first step in the process of implementing our diagonalization theorem (Theorem 1) in the milieu of computations and complexity measures. The complexity measure is still not specified, so the constructed diagonalizer can possibly be applied to any Blum-like measure.

Definition 2. *Let S be a recursive set of programs (of machines in question). Let R be the set of program codes, $R =_{df} \{1^k0^l \,|\, k, l \in N;\, k, l > 0;\, bin(k) \in S\}$ (where $bin(k)$ is the binary code of k). Let F be a mapping, $F : R \to S$, $F(1^k0^l) = bin(k)$. For any $p \in S$, let L_p be a uniformly recursive part of $L(M_p)$.*

Then by diagonalizer *M we mean the machine that works on the input of length n as follows: M first checks whether the input is of the form $1^k0^l1^j$, $k, l, j \in N$, $k, l > 0$, and $j \geq 0$. Then M constructs the initial segment of its worktape of length $\log n$, $n = k+l+j$. Within this segment, M constructs $bin(k)$ and tries to verify that $bin(k) \in S$. If $bin(k) = p \in S$ then $1^k0^l = r \in R$ and M tries further to decide whether $1^k0^l \in L_p$ (i.e. whether $r \in L_{F(r)}$) using only the initial segment of the worktape of length $\log n$ constructed previously by M. If M accomplishes this decision, then M accepts iff $1^k0^l \notin L_p$ ($r \notin L_{F(r)}$). Otherwise, M simulates p on longer input $1^k0^l1^{j+1}$ in the same manner as universal machine U can do. (This simulation is not limited in the amount of used tape.)*

Moreover, for $r \in R$ we define $z(r)$ to be the minimal j such that working on $r1^j$, diagonalizer M decides whether $r \in L_{F(r)}$ or not.

This definition has introduced a diagonalizer which is appropriate for language separation tasks. A similar diagonalizer can be defined which is appropriate for proving separation results for unary languages. The following theorem translates Theorem 1 into the world of computations and Blum-like complexity measures.

Theorem 3. *Let m be a measure and c be a complexity bound. Let S, R, F, L_p, M, z be as in Definition 2, and $C =_{df} \{L_p \,|\, p \in S\}$. For each $r \in R$, let the following two conditions hold:*
a) $r1^{z(r)} \in L_{m,p_M,c(n+1)} \leftrightarrow r \notin L_{F(r)}$,
b) for each $j < z(r)$ $(r1^j \in L_{m,p_M,c(n+1)} \leftrightarrow r1^{j+1} \in L_{F(r)})$.
Then $L_{m,p_M,c(n+1)} \notin \mathcal{E}(C)$.

Proof. S, R, F, C satisfy the assumptions of Theorem 1. It is clear that also z and $L = L_{m,p_M,c(n+1)}$ satisfy the assumptions of Theorem 1 and especially its conditions (a) and (b). The statement follows immediately. □

6 The Separation Result for the Distance Measure

We introduce a new complexity measure which we call the *distance complexity.* Let d, $d : N \to N$, be a positive function. For any machine, we define the d-subsequence $\{c_i\}$ of its computation on a word u in question as follows:

1. c_0 is the initial configuration.
2. Given c_i, the next configuration c_{i+1} is defined to be the first configuration c after c_i such that there is a configuration c' in between c_i and c (along the computation) or $c' = c_i$, and the distance between the worktape head positions of c and c' equals $d(|u|)$.

The distance complexity measure m^d is defined as follows. For $u \in \{0,1\}^*$ and for $p \in S$, define $m^d(u, p, \epsilon)$ to be the minimum length of the d-subsequences over all accepting computations of M_p on u. Clearly, the *buffering complexity* is obtained as a special case of the distance measure when we demand $c' = c_i$ in part 2 of the definition of d-subsequence above.

Note that by means of the length of subsequences one can also define the classical time or space complexity measures. The subsequence is simply the whole computation for the time complexity while the space complexity is obtained when the subsequence contains exactly each configuration c_k such that the worktape head position of any previous configuration c_i, $i < k$, is to the left of the worktape head position of c_k.

Lemma 4. *Let d be a function, U be the universal machine, and u be an input word. Then $m^d(u, p_M, \epsilon) = m_U^{d+|p_M|}(u, p_M)$.*

Proof. Hint. On the worktape of the universal machine U the key property of the simulation of M is that the program p_M is being shifted along the tape following the shifts of the head of M. The distance $d(n)$ on the worktape of M is transformed to the distance $d(n) + |p_M|$ on the worktape of U. □

Now we prove the separation result for the distance complexity measure.

Theorem 5. *Let U be a fixed universal machine, c be a recursive complexity bound, and d, $d : N \to N$, be a recursive nondecreasing function satisfying $d > \log_2$. Then there is a language $L \in C_{m_U^{d(n+1)+K}, c(n+1)}$ (where K is a constant) which is not in the class $\mathcal{E}(C_{m_U^d, c})$.*

Proof. Let S, R, F, L_p, M, z be as in Definition 2 and $L_p =_{df} L_{m_U^d, p, c}$. Let C be also as in Definition 2, $C = C_{m_U^d, c}$, and $m =_{df} m^{d(n+1)}$. We define $L =_{df} L_{m^{d(n+1)}, p_M, c(n+1)}$.

We will prove that $L \notin \mathcal{E}(C)$. It suffices to verify conditions (a) and (b) of Theorem 3. For $r \in R$, machine M, working on $r1^{z(r)}$, uses only $\log n < d(n)$ cells of its worktape. Hence, M decides, whether $r \in L_{F(r)}$ or not, within the complexity bound 1 corresponding to the initial configuration in the d-subsequence ($\log n < d(n)$) under the measure $m = m^{d(n+1)}$. Thus, condition (a) is satisfied.

For $j < z(r)$, let us verify that $r1^j \in L_{m^{d(n+1)}, p_M, c(n+1)} \leftrightarrow r1^{j+1} \in L_{m_U^d, F(r), c}$ $= L_{F(r)}$. This is true since from the definition of M, it follows that, in this case, M works as U does. Hence, condition (b) of Theorem 3 holds. According to Theorem 3, we have $L \notin \mathcal{E}(C)$.

Furthermore,

$$\begin{aligned} L &= L_{m^{d(n+1)},p_M,c(n+1)} \\ &= \Big\{ u \in \{0,1\}^* \;\Big|\; m^{d(n+1)}(u,p_M,\epsilon) \le c(|u|+1) \Big\} \\ &= \Big\{ u \in \{0,1\}^* \;\Big|\; m_U^{d(n+1)+|p_M|}(u,p_M) \le c(|u|+1) \Big\} \quad \text{(according to Lemma 4)} \\ &= L_{m_U^{d(n+1)+|p_M|},p_M,c(n+1)} \in C_{m_U^{d(n+1)+|p_M|},c(n+1)} . \end{aligned}$$ □

7 The Hierarchy

In order to prove the hierarchy theorem, we first show the folowing lemma.

Lemma 6. *Let c_1, c_2 be complexity bounds and d_1, d_2 be functions, $d_1 : N \to N$, $d_2 : N \to N$. If $c_1 \le c_2$ and $d_1 \le d_2$, then $C_{comp,m_U^{d_1},c_1} \subseteq C_{comp,m_U^{d_2},c_2}$.*

Proof. Let $L \in C_{comp,m_U^{d_1},c_1}$. Then there is a program $p \in S$ such that $L = L_{m_U^{d_1},p,c_1} = L(M_p)$. It suffices to prove that $L_{m_U^{d_1},p,c_1} \subseteq L_{m_U^{d_2},p,c_2}$ which implies $L = L_{m_U^{d_2},p,c_2} = L(M_p)$, and consequently $L \in C_{comp,m_U^{d_2},c_2}$. Suppose $u \in L_{m_U^{d_1},p,c_1}$.

For $i = 1, 2$, let c^i_j be the j-th configuration of d_i-subsequence of a computation of U on the input u with p given on the worktape of the starting configuration. For our purposes, it suffices to prove that for each j, c^1_j is not later in the computation than c^2_j. We know that $c^1_0 = c^2_0$. We continue by contradiction. Let j be the first number that the configuration c^1_j is not after c^2_j but c^1_{j+1} is after c^2_{j+1}. Thus in the sequence of configurations between c^2_j and c^2_{j+1} we find a configuration whose worktape head position is within the distance of $d_2(n) \ge d_1(n)$ from the worktape head position of c^2_{j+1} which contradicts the assumption that c^1_{j+1} is after c^2_{j+1}. Hence, $u \in L_{m_U^{d_2},p,c_2}$. □

We see that for the language L from Theorem 5, $L \notin \mathcal{E}(C_{m_U^d,c}) \supseteq C =_{df} \mathcal{E}(C_{comp,m_U^d,c})$ holds. In order to obtain a hierarchy we search for $d_1 \ge d$ and $c_1 \ge c$ such that $L \in C_1 =_{df} C_{comp,m_U^{d_1},c_1}$. According to Lemma 6, this will give $C \subseteq C_1$ for complete classes C and C_1, which together with $L \in C_1 \setminus C$ will provide the desired hierarchy.

Recall from the proof of Theorem 5 that $L = L_{m^{d(n+1)},p_M,c(n+1)}$. We will define machine M' such that $L(M') = L$. We will first describe the main ideas of how M' computes. At the beginning of its computation, M' constructs on its worktape two segments of lengths $4\log(d(n+1))$ and $\log(c(n+1))$, respectively. Then M' simulates M so that M' shifts the two segments along the worktape together with the worktape head of M. The first segment of length $4\log(d(n+1))$ serves for identifying the time instant at which the current configuration of M belongs to the $d(n+1)$-subsequence. For this purpose, it suffices to keep and update the current head position, the minimum and maximum head positions—all these three positions measured as a (possibly oriented) distance from the head position of the previous-element configuration from the $d(n+1)$-subsequence. In addition, this includes a test whether a current maximum distance between two head positions equals $d(n+1)$ which requires the value $d(n+1)$ to be precomputed and shifted within this first segment. Hence, the length of the first segment follows. Similarly, the second segment of length $\log(c(n+1))$ serves for halting the computation after $c(n+1)$ configurations of the $d(n+1)$-subsequence.

In fact, the implementation of the ideas above requires a slightly longer segments due to the fact that the worktape alphabet of M' is restricted to $\{0,1\}$ according to our definition of Turing machine. In particular, it suffices to replace each bit by a pair of bits. The first bit of this pair indicates "marked/non-marked" which is used e.g. for comparing two parts of segments, and the second one represents the value of the original bit. So, the introduced segments must be twice as long as above.

Obviously, $L(M') = L$. Moreover, $L(M') \in C_{comp,m_U^{d'},c'}$ for

$$d' = d(n+1) + 8\log(d(n+1)) + 2\log(c(n+1)) + K\,, \tag{1}$$
$$c' = c(n+1) + D \tag{2}$$

where $K = |p_{M'}|$ is a constant and $D : N \to N$ is a function of n which compensates for the consumption of the source for constructing the segments and computing the values of $d(n+1)$ and $c(n+1)$.

The hierarchy result is summarized in the following theorem.

Theorem 7. *Let U be a fixed universal machine, c be a recursive complexity bound, and d, $d : N \to N$, be a recursive nondecreasing function satisfying $d > \log_2$. Let functions $c' : N \to N$ and $d' : N \to N$ be defined by formula (2) and (1), respectively. Then $C_{comp,m_U^d,c} \subsetneqq C_{comp,m_U^{d'},c'}$.*

8 Conclusions

In this paper we have introduced a new distance complexity measure for computations on Turing machines with one worktape. This measure can be used for investigating the buffering aspects of Turing computations. As a first step along this direction, we have proven quite strong separation and hierarchy results. The presented theorems can even be strengthened to unary languages (by modifications in Definition 2 of diagonalizer M). Many questions concerning e.g. the comparison to other complexity measures, reductions, completeness and complexity classes remain open for further research.

We have also formulated our diagonalization method for general Blum-like complexity measures which is interesting by its own. Analogous theorems can possibly be proven for other types of machines such as those with auxiliary pushdown or counter, or with oracle etc.

References

1. Allender, E., Beigel, R., Hertrampf, U., Homer, S.: Almost-everywhere complexity hierarchies for nondeterministic time. Theoretical Computer Science 115(2), 225–241 (1993)
2. Blum, M.: A machine-independent theory of the complexity of recursive functions. Journal of the ACM 14(2), 322–336 (1967)
3. Cook, S.A.: A hierarchy for nondeterministic time complexity. Journal of Computer and System Sciences 7(4), 343–353 (1973)
4. Geffert, V.: Space Hierarchy Theorem Revised. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, pp. 387–397. Springer, Heidelberg (2001)
5. Kinne, J., van Melkebeek, D.: Space hierarchy results for randomized models. In: Albers, S., Weil, P. (eds.) Proceedings of the STACS 2008 Twenty-Fifth Annual Symposium on Theoretical Aspects of Computer Science. LIPIcs, vol. 1, pp. 433–444. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2008)
6. Seiferas, J.I.: Relating refined space complexity classes. Journal of Computer and System Sciences 14(1), 100–129 (1977)
7. Seiferas, J.I., Fischer, M.J., Meyer, A.R.: Separating nondeterministic time complexity classes. Journal of the ACM 25(1), 146–167 (1978)
8. Sudborough, I.H.: Separating tape bounded auxiliary pushdown automata classes. In: Mitzenmacher, M. (ed.) Proceedings of the STOC 1977 Ninth Annual ACM Symposium on Theory of Computing, pp. 208–217. ACM (1977)
9. Žák, S.: A turing machine time hierarchy. Theoretical Computer Science 26, 327–333 (1983)

Author Index